NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

TESTING AN IMPLEMENTATION'S CONFORMANCE TO A FORMAL SPECIFICATION: THE SNR HIGH SPEED TRANSPORT PROTOCOL

by

Robert Baxter Grier, Jr.

March 1995

Thesis Advisor: G.M. Lundy

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE March 1995	3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE Testing an Implementation's Conformance to a Formal Specification: The SNR High Speed Transport Protocol		5. FUNDING NUMBERS
6. AUTHOR(S) Grier, Robert Baxter, Jr.		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/ MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.		
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The major problem addressed by this research is testing the actual implementation of a high speed networking transport protocol, SNR, written by two masters degree candidates, Wan and Mezhoud, to determine its adherence to a formal specification described by H. A. Tipici and G. M. Lundy.

The approach taken was to modify the code to provide a program trace which included information about internal state variables and was designed to follow the specification's finite state machine description. The specification was used in conjunction with Testgen, a program written by C. Basaran, to generate a set of verification tests. A program was designed and implemented to provide a detailed analysis of the implementation, based on these two sets of data, to identify any deviations from the specification.

The results of this work found machines T2, R1 and R2 perform the dequeuing of packets in unspecified states, and that R4 fails to check for an empty INBUF before finishing. The automated verification process enabled the detailed inspection of hundreds of lines of trace listings in seconds, providing information about which transitions were actually taken and error messages when failures to perform required actions occurred or predicate requirements were not met.

14. SUBJECT TERMS Transport Protocol, Networking, Software Testing			15. NUMBER OF PAGES 80
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

TESTING AN IMPLEMENTATION'S CONFORMANCE TO A FORMAL SPECIFICATION: THE SNR HIGH SPEED TRANSPORT PROTOCOL

Robert Baxter Grier, Jr.
Captain, United States Army
B.S., Texas A&M University, 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1995

Accesion For	
NTIS CRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

Author: [illegible]
Robert Baxter Grier, Jr.

Approved by: [illegible]
G. M. Lundy, Thesis Advisor

[illegible]
Shridhar B. Shukla, Second Reader

Ted Lewis
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The major problem addressed by this research is testing the actual implementation of a high speed networking transport protocol, SNR, written by two masters degree candidates, Wan and Mezhoud, to determine its adherence to a formal specification described by H. A. Tipici and G. M. Lundy.

The approach taken was to modify the code to provide a program trace which included information about internal state variables and was designed to follow the specification's finite state machine description. The specification was used in conjunction with Testgen, a program written by C. Basaran, to generate a set of verification tests. A program was designed and implemented to provide a detailed analysis of the implementation, based on these two sets of data, to identify any deviations from the specification.

The results of this work found machines T2, R1 and R2 perform the dequeuing of packets in unspecified states, and that R4 fails to check for an empty INBUF before finishing. The automated verification process enabled the detailed inspection of hundreds of lines of trace listings in seconds, providing information about which transitions were actually taken and error messages when failures to perform required actions occurred or predicate requirements were not met.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

A. BACKGROUND

Today, computer networking and networking issues and problems are a major area of interest in the field of computer science. Computers from desktop PCs to workstations and mainframes are being linked together in order to share resources from printers, to memory, to processor power. As the processing speed of these networked machines has increased, so has the requirement for intercommunication. Hardware designed to allow connectivity between separate machines has grown at an astounding rate with the advent of fiber optic cable and network standards such as FDDI and fast Ethernet. These new network standards, capable of data transfer rates greater than 100 million bits per second, have outstripped existing transport protocols. The problem of developing new transport protocols to better utilize the vast bandwidths available with these new standards has given rise to numerous designs from both the commercial and academic worlds.

Protocol designs must first be described in a formal specification. The formal specification needs to allow for both a verification of the protocol and should be complete enough to translate directly into an actual implementation.

One such answer to meeting the needs of high speed networks is the SNR transport protocol. Originally described in [1] and further refined in [10], this protocol is sufficiently well defined to be implemented. The nature of this work is to take an actual implementation and walk it against the formal specification to verify compliance.

As with any piece of software, the correctness and viability of these new protocols must be carefully examined and tested. Testing must be done for the formal specification and for the implementation as well. A number of formal testing procedures for the specification of a networking protocol have been developed to include reachability analysis. Testing an implementation to ensure its conformance to the formal specification is another necessary requirement. The derivation of a set of finite state machines to incrementally examine and test the protocol must be done. Once the set of tests are developed, the process of applying the tests to an actual implementation may be undertaken. The generation of the test sequences must be carefully examined in order to preclude spurious results. The actual development of a testing program to examine the workings of the transport protocol implementation at work is the problem at hand.

B. OBJECTIVE

The objective of this thesis is to report the results of testing an implementation of the SNR protocol, written by F. Mezhoud [13] and W. J. Wan [12], to ensure its compliance with the formal specification. The implementations are the thesis work of two Naval Postgraduate School master's students and is run on an FDDI network consisting of two Sun Microsystems SPARCstationTM 10 running the SolarisTM 2.3 operating system.

C. SCOPE, LIMITATIONS AND ASSUMPTIONS

The scope of this work limited to designing a test program which will allow the user to examine the program traces generated by eight different SNR machines and compare it to a set of test sequences derived from the protocol specification. The goal is to determine if the implementation conforms to the specification. The SNR implementation code was augmented so state variable information is written to a file during run time. This program trace information is then compared with the formal specification in order to determine if the program is working in accordance with the design specification. This work serves only to test for compliance with specification. Each machine is tested separately and treated as a black box, that is, the inner workings of the machines are not examined, only the state of the global and local variables. This work is not intended to test if the specification contains errors or to evaluate the protocol's ability to provide service to the user.

D. ORGANIZATION OF THESIS

This thesis is organized into five chapters. This chapter is the introduction and serves to introduce the reader to the problem of transport protocol implementation specification compliance testing, the purpose of this work. Chapter II gives an overview of the SNR protocol along with a review of the Systems of Communicating Machines (SCM) which serves to model the protocol. Chapter II also contains information about the modifications made to the specification in order to facilitate the testing process. Chapter III discusses the underlying concepts behind the test generation process, the method of producing both test traces from the implementation and test sequences from the specification, and details concerning the automated verification programs. Chapter IV discusses the results obtained from the examination of the implementation. Chapter V contains conclusions which may be derived from the given results and suggests additional work which remains to be done.

II. THE SNR PROTOCOL

A. BACKGROUND

The SNR transport protocol is an attempt to overcome the difficulties experienced by the current transport protocols with some unique features which are different than the features of the other lightweight protocols. It was first introduced in [1], and in [10] a formal specification was given by using the Systems of Communicating Machines (SCM) model. The next section defines the protocol model SCM.

B. SYSTEMS OF COMMUNICATING MACHINES (SCM)

A number of models for protocol specification and verification have been defined; these are discussed in the references [3], [4], [5], [6], [7], and [8]. The model used here is called *systems of communicating machines*, and is briefly described in this section. A more detailed description and discussion can be found in references [9] and [10].

A *system of communicating machines* is an ordered pair $C=(M, V)$, where

$M=\{m_1, m_2,\ldots, m_n\}$

is a finite set of *machines*, and

$V=\{v_1, v_2,\ldots, v_k\}$

is a finite set of *shared variables*, with two designated subsets R_i and W_i specified for each machine m_i. The subset R_i of V is called the set of *read access variables* for machine m_i, and the subset W_i the set of *write access variables* for m_i.

Each machine $m_i \in M$ is defined by a tuple $(S_i, s_0, L_i, N_i, \tau_i)$, where

(1) S_i is a finite set of states;

(2) $s_0 \in S_i$ is a designated state called the *initial state* of m_i;

(3) L_i is a finite set of *local variables*;

(4) N_i is a finite set of names, each of which is associated with a unique pair (p, a), where p is a *predicate* on the variables of $L_i \cup R_i$ and a is an *action* on the variables of

$L_i \cup R_i \cup W_i$. Specifically, an action is a partial function a:$L_i \times R_i \rightarrow L_i \times W_i$ from the values contained in the local variables and read access variables to the values of the local variables and write access variables.

(5) τ_i: $S_i \times N_i \rightarrow S_i$ is a transition function, which is a partial function from the states and names of m_i to the states of m_i.

Let $\tau(s_1, \text{n}) = s_2$ be a transition which is defined on machine m_i. Transition τ is *enabled* if the enabling predicate p, associated with name n, is true. Transition τ may be executed whenever m_i is in state s_1 and the predicate p is true (enabled). The *execution* of τ is an atomic action, in which both the state change and the action a associated with n occur simultaneously.

The set L_i of local variables specifies a name and a range for each. The range must be a finite or countable set of values.

A table called a *predicate-action table (PAT)* lists each transition name and the predicate and action associated with that transition. This table, together with the FSM diagrams and the variables make up the formal specification.

C. THE SNR TRANSPORT PROTOCOL

The following abstract communication structure definitions are from [1]. The majority of the following text is extracted from [10] with some modifications.

1. Design Philosophy

The key idea in the design of the SNR protocol is to provide rapid processing of packets. This goal is achieved through simplicity, reduction of processing overhead and utilization of parallel execution of tasks. In order to achieve these goals, the following design principles are observed:

- Periodic exchange of complete state information and eliminating explicit timers,
- Selective repeat method of retransmission,
- The concept of packet blocking,
- Parallel processing.

Further elaboration of these design principles can be found in [1] and [10].

2. Modes of Operation

The following three modes of operation are specified:

Mode 0 has no error control or flow control. It is suited for virtual circuit networks and for the cases where quick interaction between the communicating entities is desired and short packets are used.

Mode 1 has no error control but provides flow control. This mode is suitable for real time applications such as packetized voice or real-time monitoring of a remote sensor where error control is not needed and packet sizes are small. Also convenient if the underlying network is reliable.

Mode 2 has both error control and flow control. This is the most reliable mode and it is useful for large file transfers in all types of network services.

3. Machine Organization and General Overview

The protocol can be envisioned as connecting two host computers end-to-end across a high speed network as shown in Figure 1.

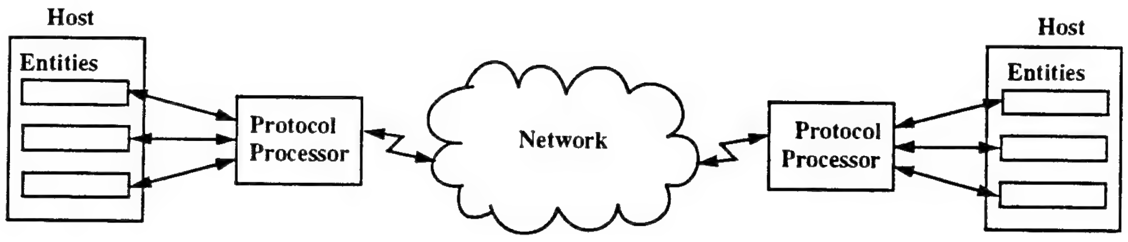


Figure 1 : Network, Hosts, Entities and Protocol Processors

This protocol requires a full duplex link between two host systems. Each host system in the network consists of eight finite state machines (FSM), four for executing the transmitter functions, and four for executing the receiver functions.

The general organization of the machines is shown in Figure 2 (this figure is an extension of a similar figure in [1]). Each machine in the protocol performs a specific function in coordination with other machines. The coordination is established by communicating through *shared variables* which will be explained later.

Machine T1 is responsible for the transmission of new data packets and retransmission of old packets. Machine T2 establishes the connection with the receiver and thereafter processes the incoming receiver control packets and updates related tables and variables as the blocks are acknowledged. Machine T3 sends transmitter control packets to the receiver periodically. Machine T4 is the host interface of the transmitter. It inserts the incoming data stream into the buffer for transmission by machine T1.

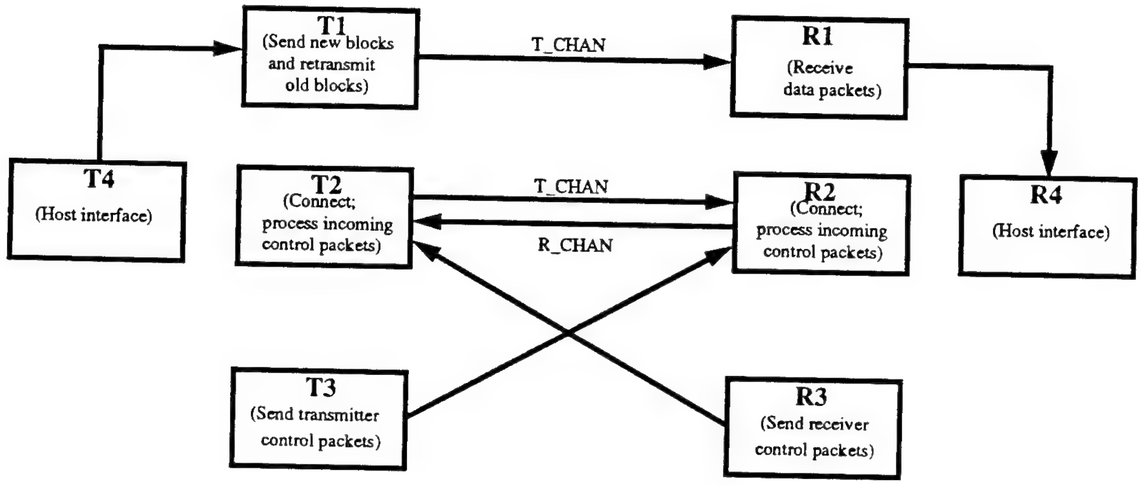

T1
(Send new blocks and retransmit old blocks)
T_CHAN
R1
(Receive data packets)
T4
(Host interface)
T2
(Connect; process incoming control packets)
T_CHAN
R_CHAN
R2
(Connect; process incoming control packets)
R4
(Host interface)
T3
(Send transmitter control packets)
R3
(Send receiver control packets)

Figure 2 : Machine Organization

Machine R1 removes the data packets from the transmitter channel and inserts them into the buffer in order according to their sequence numbers. Machines R2 and R3 are receiver counterparts of transmitter machines T2 and T3. Machine R2 receives the connection request messages sent by machine T2. After the connection establishment, it receives the transmitter control packets. Machine R3 sends the receiver control packets at periodic intervals through the receiver channel. Machine R4 is the host interface of the receiver. It retrieves the data packets from the buffer and passes them to the host.

4. Services Provided

The protocol provides for the following general services:

Multiplexing, demultiplexing,

Connection management,

Sequenced delivery,

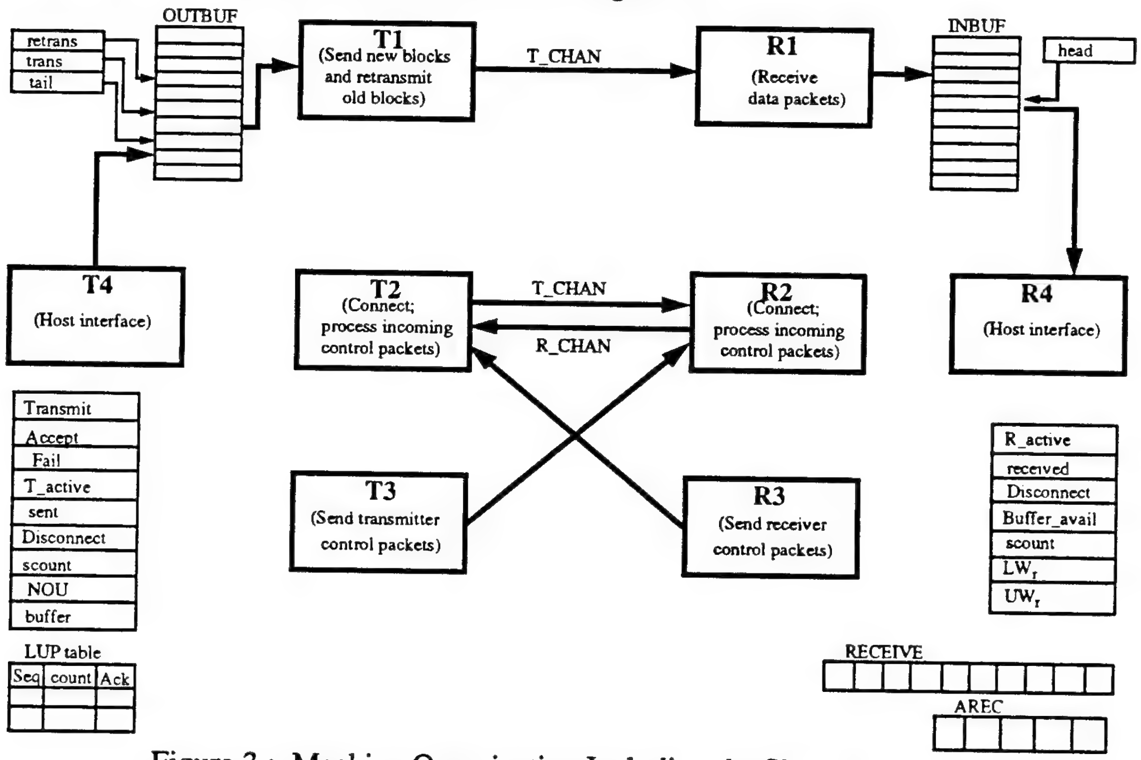
Flow control,

Error recovery.

D. COMMUNICATION STRUCTURES

In this section the communication structures are defined for background information. The information provided it taken directly from [10].

To illustrate discussions in this section the machine organization diagram, Figure 2, is extended to include communication structures as well as the global shared variables in the transmitter and the receiver and shown as Figure 3.



OUTBUF
retrans
trans
tail
T1 (Send new blocks and retransmit old blocks)
T_CHAN
R1 (Receive data packets)
INBUF
head
T4 (Host interface)
T2 (Connect; process incoming control packets)
T_CHAN
R_CHAN
R2 (Connect; process incoming control packets)
R4 (Host interface)
Transmit
Accept
Fail
T_active
sent
Disconnect
scount
NOU
buffer
T3 (Send transmitter control packets)
R3 (Send receiver control packets)
R_active
received
Disconnect
Buffer_avail
scount
LW_r
UW_r
LUP table
Seq
count
Ack
RECEIVE
AREC

Figure 3 : Machine Organization Including the Shared Variables

1. Communication Channels

The logical links connecting the two entities are modeled as queues which are called "communication channels" in the specification of the protocol.

T_CHAN is the channel from the transmitter to the receiver. This is the channel in which the transmitter sends connection requests, connection confirmation messages, data packets, transmitter control packets and disconnect messages. As shown in Figure 3, T1 and T2 write messages into T_CHAN, while R1 and R2 read messages from T_CHAN.

R_CHAN is the channel from the receiver to the transmitter. This channel carries the connection acknowledgment messages and receiver control packets sent by the receiver (R2 and R3).

2. Buffers

Figures 4 and 5 show the buffers used in the transmitter and the receiver. It is assumed that the data stream is already divided into packets by the host.

OUTBUF: This is where machine T4 deposits the data packets it gets from the host for transmission. Machine T1 extracts the packets from here, adds the header parts and transmits them to the receiver. A schematic illustration of *OUTBUF* is shown in Figure 4.

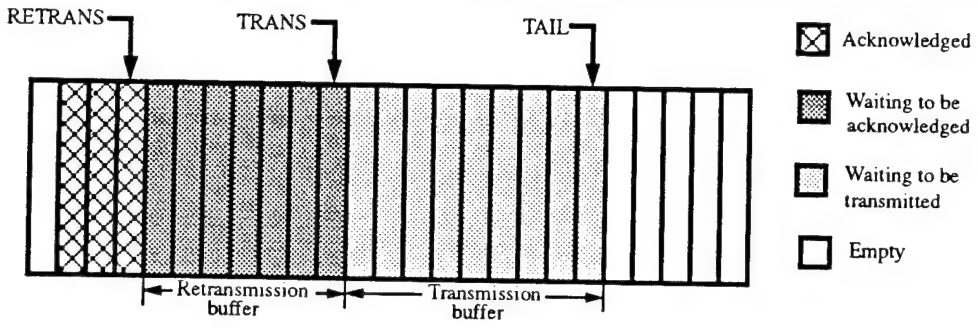


Figure 4 : *OUTBUF*: Transmitter Buffer

As it can be seen in the figure, this buffer has two parts: Retransmission buffer and transmission buffer. The retransmission buffer is located before the transmission buffer and holds the packets that have been transmitted by machine T1 but not acknowledged yet. The transmission buffer holds the packets which have been buffered by machine T4 and waiting to be transmitted. These buffers are marked by three pointers called *RETRANS*, *TRANS* and *TAIL*.

The purpose of dividing the *OUTBUF* into two parts is to avoid the movement of data packets within the buffer, which is a costly operation. With the buffer scheme explained here, the data packets which are enqueued at the end of the transmission buffer remain in their places until they are acknowledged by the receiver.

INBUF: Each logical connection has its own pre-negotiated buffer in the receiver called *INBUF*. This temporarily holds the data packets until they are retrieved by the receiving host. Another function of this buffer is to reorder the data packets that arrive out of sequence. Received data packets are inserted into buffer locations whose indexes are calculated from the sequence numbers. Figure 5 depicts the schematic diagram of *INBUF*.

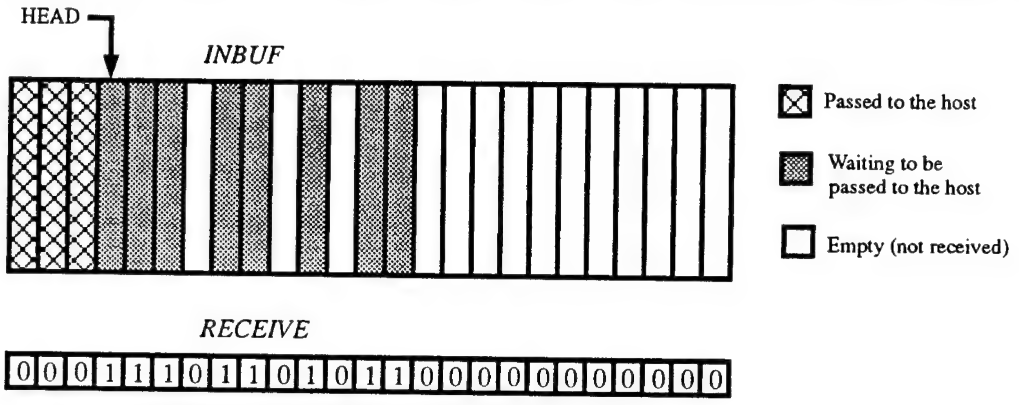


Figure 5 : *INBUF* and *RECEIVE*: Receiver Buffers

RECEIVE: This is an array of bits where each bit maps to a location of *INBUF*. The purpose of this bit array is to indicate if any location of *INBUF* contains a data packet. A *RECEIVE* bit set means that there is data in the corresponding *INBUF* location. This scheme has three uses: First, it helps detection of duplicate packets whose block sequence numbers are greater than LW_r, secondly, it is used in determining whether or not a whole block has been received for acknowledgment purposes and finally it indicates to machine R4 whether there is a data packet in the buffer ready to be retrieved.

After machine R4 passes a block of packets to the host, it sets the corresponding *RECEIVE* bits to 0. The buffer allocation and deallocation for the packets is done in a very simple way by the protocol and no operating system support is needed, except for the allocation of a buffer space for *INBUF* in the memory.

AREC: This is another array of bits, whose size is equal to the number of blocks that can be stored in *INBUF*. Each bit in this array corresponds to a block of packets in *INBUF* starting from the first location, so that bit 1 of *AREC* corresponds to the first block of packets, bit 2 corresponds to the second block of packets and so on. When all the packets in a block have been received, the *AREC* bit for this block is set to 1. This array is used to acknowledge the blocks together with LW_r and *LOB* array.

These structures are used as follows in order to acknowledge the data packets received correctly: Upon reception of a data packet by the receiver, a check for duplicate detection is done. The packet is a duplicate if the block number that contains the packet is less than LW_r, or if the *RECEIVE* bit corresponding to the packet sequence number is 1. In this case, the packet is discarded. Otherwise, it is inserted into the corresponding *INBUF* location and the *RECEIVE* bit for this location is set to 1. If this packet completes the reception of a whole block of packets, then either LW_r is increased until it is equal to the sequence number of the first incomplete block (if the completed block is LW_r), or a bit corresponding to that block in *AREC* is set to 1 (if the completed block is different than LW_r). Thereafter, *AREC* is copied into the *LOB* array for transmission in a receiver control packet to acknowledge the successfully received blocks.

3. Major Variables

The major transmitter variables are shown in Table 1 and the major receiver variables in Table 2.

name	accessed by machines	type or purpose
OUTBUF	T1,T4	buffer messages for transmission
retrans,trans,tail	T1 (subroutines)	pointers to OUTBUF
Transmit	T2,T4	boolean
Accept	T2,T4	boolean
Fail	T2,T4	boolean

Table 1: Major Transmitter Variables

name	accessed by machines	type or purpose
T_active	T1,T3,T4	boolean
sent	T1,T3	boolean
Disconnect	T2,T3	boolean
scount	T2,T3	counter
NOU	T1,T2	number of unack'd data blocks
LUP(seq,count,ack)	T1,T2	table for transmitted data blocks
mode	T1,T2	store current mode
UW_t	T1	upper window, transmitter
LW_t	T2	lower window, transmitter
k	T2,T3	interval between control packet transmissions
L	T1	max window size

Table 1: Major Transmitter Variables

name	accessed by machines	type or purpose
INBUF	R1,R4	store incoming data packets
head	R1,R4	pointer to INBUF
R_active	R1,R2,R3,R4	boolean
Disconnect	R3,R4	boolean
Buffer_avail	R1,R4	boolean
scount	R2,R3	counter
LW_r	R1,R4	lower window, receiver
UW_r	R1,R4	upper window, receiver
RECEIVE	R1,R4	bit array for packets in INBUF
AREC	R1,R4	bit array, blocks in INBUF
mode	R1,R4	mode indicator, 0,1 or 2
LOB	R1,R4	bit array, for acknowledgments
k	R3	interval between control packet transmissions
received	R1,R3	boolean

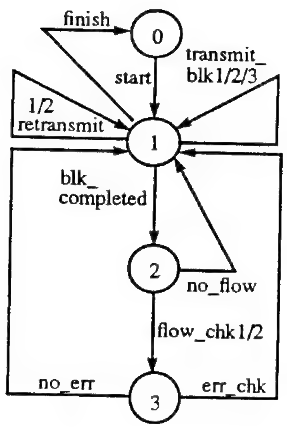
Table 2: Major Receiver Variables

E. FORMAL SPECIFICATION--MODIFIED FOR TESTING

In this section, a formal specification of the SNR transport protocol will be given using the SCM model. The FSMs and PATs given are the ones used to define the specification for the tests. No new states were added to those given in [10] but some transitions were added. The additional transitions are the result of removing all the *or* conditions in the predicates and the conditional statements from the actions. This gives a set of predicates and actions which may be directly compared with the test run trace data.

Another approach to removing conditional predicates would have been to add additional states, transitional in nature, exited based on the conditional portions of the original transitions. This approach is less preferable as it increases the complexity of both the FSM and the PAT. Where n is the number of *or* conditions in the original specification, adding states increases the transition number by *n* versus *n-1* for the splitting method.

1. Modifications to the Transitions



Transition	Predicate	Action
start	T_active=T	null
finish	T_active=F	null

retransmit1	T_active=T ^
	mode= 2 ^ **Expired**(LUP) /=0
	^ retrans_count <= block_size
		Packet.seq:=(**Expired**(LUP)-1)*block_size+
		retrans_count;
		Packet.data:=OUTBUF(Packet.seq mod
		OUTBUF'length);
		Enqueue(Packet,T_CHAN);
		sent := T;
		inc (retrans_count);

retransmit2	T_active=T ^
	mode= 2 ^ **Expired**(LUP) /=0
	^ retrans_count > block_size
		Packet.seq:=(**Expired**(LUP)-1)*block_size+
		retrans_count;
		Packet.data:=OUTBUF(Packet.seq mod
		OUTBUF'length);
		Enqueue(Packet,T_CHAN);
		sent := T;
		retrans_count:=1;
		LUP((**Expired**(LUP)-1) mod L+1).count :=
		initial value;

transmit_
blk1
	T_active=T ^
	not (**Empty**(OUTBUF)) ^
	trans_count <= blk_size ^
	mode=0
		retrans_count := 1;
		Packet.seq:=UW_t * block_size + trans_count;
		Dequeue(Packet.data,OUTBUF);
		Enqueue(Packet,T_CHAN);
		sent:=T;
		inc (trans_count);

transmit_
blk2
	T_active=T ^
	not (**Empty**(OUTBUF)) ^
	trans_count <= blk_size ^
	NOU < L ^
	buffer - NOU>0 ^
	mode=1
		retrans_count := 1;
		Packet.seq:=UW_t * block_size + trans_count;
		Dequeue(Packet.data,OUTBUF);
		Enqueue(Packet,T_CHAN);
		sent:=T;
		inc (trans_count);

transmit_
blk3
	T_active=T ^
	not (**Empty**(OUTBUF)) ^
	trans_count <= blk_size ^
	NOU < L ^
	buffer - NOU>0 ^
	Expired(LUP)=0
		retrans_count := 1;
		Packet.seq:=UW_t * block_size + trans_count;
		Dequeue(Packet.data,OUTBUF);
		Enqueue(Packet,T_CHAN);
		sent:=T;
		inc (trans_count);

blk_
completed
	trans_count > blk_size	trans_count := 1;
		inc (UW_t);

no_flow	mode = 0	null
flow_chk1	mode = 1	inc (NOU);
flow_chk2	mode = 2	inc (NOU);
no_err	mode = 1	null
err_chk	mode = 2	**Insert** (UW_t, LUP);

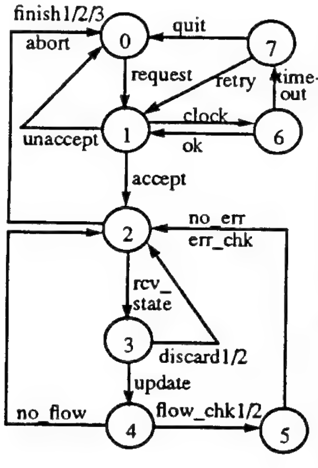
Figure 6 : T1 State Diagram and Modified Predicate-Action Table

Machine T1 is responsible for transmission of new data packets and retransmission of unacknowledged packets whenever required. Figure 6 shows the state diagram and the modified predicate-action table.

The retransmit transition was split into two, the difference being in what was previously a conditional action based on the comparison of retrans count and block size.

The transmit block transition was split into three based on the conditional portion of the original predicate conditions: mode=0, or mode=1 and buffer - NOU > 0, or Expired(LUP) = 0 and buffer - NOU > 0.

The flow check transition was split to reflect mode 1 or mode 2.



Transition	Predicate	Action
request	Transmit=T ^ Accept=T ^ Fail=F	**Enqueue** (*Conn_Req*, T_CHAN);
accept	R_CHAN(front) = *Conn_Ack* ^ **Acceptable** (R_CHAN(front))	T_active := T; **Enqueue** (*Conn_Conf*, T_CHAN); **Dequeue** (R_CHAN);
unaccept	R_CHAN(front) = *Conn_Ack* ^ not (**Acceptable** (R_CHAN(front)))	Accept:=F; **Dequeue** (R_CHAN);
clock	**Empty** (R_CHAN) ^ *clock_tick*	inc (delay);
ok	delay < reset	null
timeout	delay = reset	inc (attempts); delay:=0;
retry	attempts < max_attempts	**Enqueue** (*Conn_Req*, T_CHAN)
quit	attempts = max_attempts	Fail := T
finish1	Transmit = F ^ **Empty** (OUTBUF) ^ Disconnect = F ^ mode = 1	T_active:=F; **Enqueue** (*Disc*, T_CHAN);
finish2	Transmit = F ^ **Empty** (OUTBUF) ^ Disconnect = F ^ mode = 0	T_active:=F; **Enqueue** (*Disc*, T_CHAN);
finish3	Transmit = F ^ **Empty** (OUTBUF) ^ Disconnect = F ^ mode = 2 ^ **Empty** (LUP)	T_active:=F; **Enqueue** (*Disc*, T_CHAN);
abort	Disconnect = T	T_active:=F; Transmit:=F;
rcv_state	not (**Empty** (R_CHAN)) ^ Disconnect=F	null
discard1	R_CHAN(front)=*Conn_Ack*	**Dequeue**(R_CHAN);
discard2	R_CHAN(front).seq <= high	**Dequeue**(R_CHAN);
update	R_CHAN(front).seq > high	scount:=0; high:=R_CHAN(front).seq;
no_flow	mode = 0	**Dequeue**(R_CHAN);
flow_chk1	mode = 1	**Balance**(R_CHAN(front).LOB,HOLD, R_CHAN(front).LW_r,LW_t, NOU); HOLD := R_CHAN(front).LOB; LW_t := R_CHAN(front).LW_r; buffer := R_CHAN(front).buffer_avail. **Update_outbuf** (OUTBUF, LW_t);
flow_chk2	mode = 2	**Balance**(R_CHAN(front).LOB,HOLD, R_CHAN(front).LW_r,LW_t, NOU); HOLD := R_CHAN(front).LOB; LW_t := R_CHAN(front).LW_r; buffer := R_CHAN(front).buffer_avail. **Update_outbuf** (OUTBUF, LW_t);
no_err	mode = 1	**Dequeue** (R_CHAN);
err_chk	mode = 2	**Update_LUP** (LUP, HOLD, LW_t, R_CHAN(front).k); **Dequeue** (R_CHAN);

Figure 7 : T2 State Diagram and Modified Predicate-Action Table

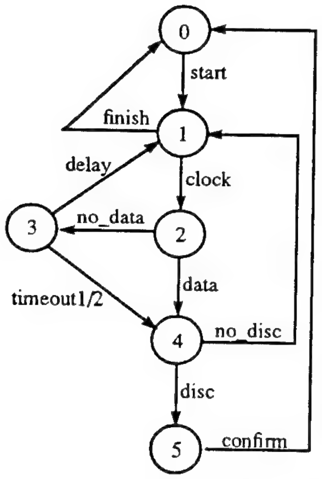
Machine T2 has two responsibilities: (*i*) connection establishment and termination, (*ii*) reception and processing of receiver control packets. The state diagram and the modified predicate action table are presented in Figure 7.

The finish transition was split into three based on the following: mode is 0, or mode is 1, or mode is 2 and Empty(LUP).

Discard was split based on R_CHAN(front).seq <= high or R_CHAN(front) = Conn_Ack.

No flow was split for mode is 1 or mode is 2.

Machine T3 has two main responsibilities in the protocol: periodic transmission of transmitter control packets and initialization of abnormal connection termination if no receiver control packets are received for a predetermined amount of time. The state diagram and the modified predicate action table are presented in Figure 8.

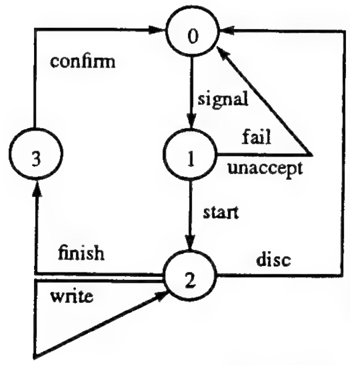


Transition	Predicate	Action
start	T_active=T	null
clock	*clock_tick* ^ T_active = T	inc (scount)
no_data	sent=F	inc (count)
delay	count < k ^ scount < Lim	null
timeout1	scount = Lim	**Enqueue** (*T_state*,T_CHAN); k:= min (2*k, klim)
timeout2	count = k	**Enqueue** (*T_state*,T_CHAN); k:= min (2*k, klim)
data	sent = T	**Enqueue** (*T_state*, T_CHAN); k:= 1
no_disc	scount < Lim	sent:= F; count:= 0
disc	scount = Lim	Disconnect:= T
confirm	T_active = F	null
finish	T_active = F	null

Figure 8 : T3 State Diagram and Modified Predicate-Action Table

The original PAT listed the predicate conditions for a timeout as count = k ***and*** scount = LIM, but the correct condition for a timeout should be count = k ***or*** scount = LIM. Timeout was split into two based on this or condition.

Machine T4 is the interface to the host transmitter and performs the necessary communication between the transmitting host and the other machines. The state diagram and the predicate action table are depicted in Figure 9.

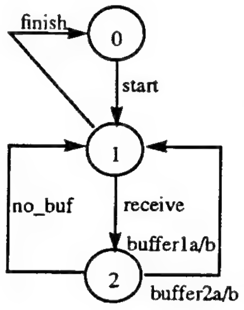


Transition	Predicate	Action
signal	*transmission signal from the host*	Transmit := T; Accept := T
fail	Fail = T	Transmit := F; *notify host of failure to connect;*
unaccept	Accept = F	*notify host of unacceptable connection*
start	T_active = T	null
write	not (**Full** (OUTBUF)) ^ not(eot) ^ T_active=T	**Enqueue** (*data stream from the host,* OUTBUF)
finish	eot ^ T_active = T	Transmit := F
confirm	T_active = F	*notify host of completion*
disc	T_active = F	*notify host of disconnect*

Figure 9 : T4 State Diagram and Predicate-Action Table

Upon receiving a transmission signal from the host, T4 initiates the execution of the protocol. As long as the connection is active, T4 writes the data into the buffer and T1 transmits them. When the end of transmission signal is received from the host, T4 initiates the connection termination and also gives necessary messages to the host informing it about the state of the connection.

Machine R1 removes the data packets from *T_CHAN* and inserts them into their



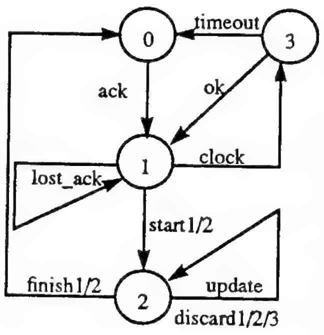
Transition	Predicate	Action
start	R_active = T	null
finish	R_active = F ^ **Empty** (INBUF)	null
receive	T_CHAN (front) = *DATA*	null
no_buf	mode = 0	*Pass* T_CHAN(front) *to the host;* **Dequeue** (T_CHAN)
buffer1a	mode = 1 ^ duplicate = F	**Order_insert**(T_CHAN(front), INBUF, RECEIVE, LW_r, duplicate); received := T; **Process_packet** (T_CHAN(front).seq, RECEIVE, AREC,Buffer_avail,LW_r,UW_r,LOB); **Dequeue** (T_CHAN);
buffer1b	mode = 1 ^ duplicate = T	**Order_insert**(T_CHAN(front), INBUF, RECEIVE, LW_r, duplicate); **Dequeue** (T_CHAN);
buffer2a	mode = 2 ^ duplicate = F	**Order_insert**(T_CHAN(front), INBUF, RECEIVE, LW_r, duplicate); received := T; **Process_packet** (T_CHAN(front).seq, RECEIVE, AREC,Buffer_avail,LW_r,UW_r,LOB); **Dequeue** (T_CHAN);
buffer2b	mode = 2 ^ duplicate = T	**Order_insert**(T_CHAN(front), INBUF, RECEIVE, LW_r, duplicate); **Dequeue** (T_CHAN);

Figure 10 : R1 State Diagram and Modified Predicate-Action Table

allocated locations in the buffer *INBUF*, discards duplicate packets, and updates the structures used for flow control and error recovery management (*RECEIVE, AREC* and

LOB). In mode 0, R1 passes the packets to the host directly without buffering and without performing any kind of error or flow control operation. The state diagram and the modified predicate action table of machine R1 are given in Figure 10.The original buffer transition had a conditional action based on whether or not the packet was a duplicate. It has been split into four, first by mode and then by the value of the duplicate flag.

Machine R2 is the receiver counterpart of transmitter machine T2. The state diagram and the predicate action table is depicted in Figure 11.



Transition	Predicate	Action
ack	T_CHAN (front) = *Conn_Req*	**Evaluate** (*Conn_req*); **Dequeue** (T_CHAN); **Enqueue** (*Conn_ack*, R_CHAN);
clock	*clock_tick* ^ **Empty** (T_CHAN)	inc (delay)
ok	delay < reset	**Enqueue** (*Conn_ack*, R_CHAN);
timeout	delay = reset	null
start1	T_CHAN(front) = *Conn_conf*	R_active:= T; **Dequeue**(T_CHAN);
start2	T_CHAN(front) = *T_state*	R_active:= T;
start3	T_CHAN(front) = *Data*	R_active:= T;
finish1	Disconnect = T	R_active := F;
finish2	T_CHAN(front)=*Disc*	R_active := F;
update	T_CHAN(front) = *T_state* ^ T_CHAN(front).seq > high	scount := 0; high := T_CHAN(front).seq; **Dequeue** (T_CHAN);
discard1	T_CHAN(front) = *Conn_conf*	**Dequeue** (T_CHAN);
discard2	T_CHAN(front) = *Conn_req*	
discard3	T_CHAN(front) = *T_state* ^ T_CHAN(front).seq <= high	**Dequeue** (T_CHAN);
lost_ack	T_CHAN(front) = *Conn_req*	**Dequeue** T_CHAN); **Enqueue** *(Conn_ack*, R_CHAN)

Figure 11 : R2 State Diagram and Modified Predicate-Action Table

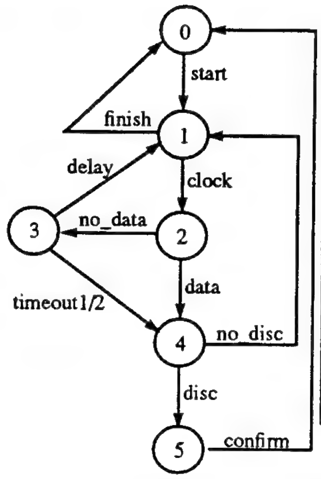
First, it establishes the connection with the transmitter and thereafter receives and processes the transmitter control packets.

The start transition was split based on the three original *or* conditions: T_CHAN(front) = *Conn_conf*, or T_CHAN(front) = *T_State*, or T_CHAN(front) = *Data*.

The finish transition split into two based on either T_CHAN(front) = *Disc* or Disconnect flag is set to true.

The discard transition was split into three: T_CHAN(front) = *Conn_conf*, or T_CHAN(front) = Conn_*req*, or T_CHAN(front) = *T_State* and T_CHAN(front).*seq* <= *high*.

Machine R3 has exactly the same structure and function as transmitter machine T3 as shown in Figure 12. The R3 PAT contained the same error as the T3, the *and* rather than *or* condition for determining timeout. Fixing this required splitting the timeout transition into two: count = k and scount = LIM.



Transition	Predicate	Action
start	R_active=T	null
clock	clock_tick ^ R_active=T	inc(scount)
no_data	received=F	inc(count)
delay	count<k ^ scount<Lim	null

timeout1	count=k	**enqueue**(R_state,R_CHAN);
		k:=min(2*k, klim)

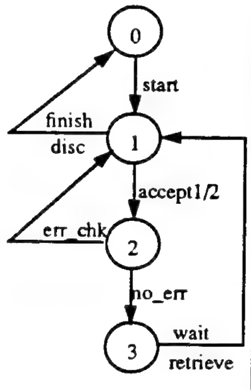
timeout2	scount=Lim	**enqueue**(R_state,R_CHAN);
		k:=min(2*k, klim)

data	received=T	**enqueue**(R_state, R_CHAN);
		k:=1

no_disc	scount<Lim	received:=F; count:=0
disc	scount=Lim	Disconnect:=T
confirm	R_active=F	null
finish	R_active=F	null

Figure 12 : R3 State Diagram and Modified Predicate-Action Table

Machine R4 provides the interface to the receiving host by passing the data in *INBUF* to the host and notifying the host of any errors which may occur during the reception of the data packets. The state diagram and the predicate-action table of machine R4 is depicted in Figure 13.The accept transition was split to reflect mode 1 or mode 2.



Transition	Predicate	Action
start	R_active = T	null

finish	R_active = F ^ **Empty** (INBUF) ^
	Disconnect = F
		null

disc	Disconnect = T	*notify host of disconnect*
accept1	Disconnect = F ^ not (**Empty**(INBUF)) ^ mode=1 ^ *signal from host*	null
accept2	Disconnect = F ^ not (**Empty**(INBUF)) ^ mode=2 ^ *signal from host*	null
no_err	mode = 1	null
wait	Wait (INBUF, RECEIVE) = T	null
retrieve	Wait (INBUF, RECEIVE) = F	**Retrieve_mode1** (INBUF,RECEIVE, AREC,buffer_avail,LW_r,UW_r,LOB);
err_chk	mode = 2	**Retreive_mode2** (INBUF,RECEIVE, buffer_avail);

Figure 13 : R4 State Diagram and Modified Predicate-Action Table

III. TESTING OF THE IMPLEMENTATION

A. TESTGEN

Testgen was written by C. Basaran and is detailed in [11].

Testgen is a program which takes as input the formal specification of a protocol using the formal model *systems of communicating machines*, and outputs a sequence of tests for an implementation of a given protocol, see Figure 14.

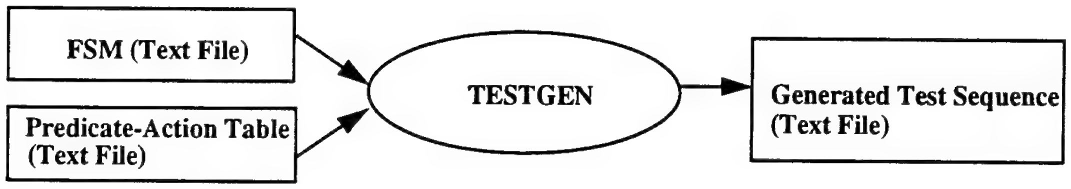

FSM (Text File)
Predicate-Action Table (Text File)
TESTGEN
Generated Test Sequence (Text File)

Figure 14 : Testgen Overview

1. A Procedure for Generating Test Sequences

In this section a procedure and its automation are described for generating a sequence of tests for a protocol specified as a SCM model. The input is the formal protocol specification (FSM and predicate-action table) specified as a *system of communicating machines* (SCM). This is a formal model, or formal description technique, defined in [14]. The output is a sequence of tests and an I/O diagram in a tabular format. The generated sequence is intended to be applied to an "inplementation under test" or IUT.

Before generating the sequence of tests and the I/O diagram for each test in the sequence, shared and local variables must be identified. The test inputs (the shared and local variables that can be set in a controlled way) and the outputs (the shared and local variables can be observed for test purposes) should be identified. These inputs and outputs form the I/O for the test steps.

The format for each single test is

$S_I\ i_1, i_2, \ldots, i_n\ ;\ o_1, o_2, \ldots, o_m\ S_E$

S_I is the state of machine when the test begins. The $i_1, i_2, \ldots, i_n$ are the values of the input variables at the start of test execution. The $o_1, o_2, \ldots, o_m$ are the values of the output variables after test execution. S_E is the state of the machine when the test is complete. The input and the output variables are taken from the shared and local variables of the machine. The determination of these variables is explained in the following section.

The testing procedure explained below is written in three parts:

- Preliminary steps,
- Test sequence generating procedure, and
- Refining steps.

2. Preliminary Steps

1. From the machine specification FSM diagram, mark each transition whose name appears on more than one transition. Each such instance for a given name is given a separate distinguishing label.
2. From the *predicate-action table*, note the number of clauses in each enabling predicate. Mark each clause. An enabling predicate may consist of several clauses, any one of which might be true, allowing the transition to execute. Marking each clause insures that each one is tested individually.
3. For each shared variable x, determine if x is an input variable, an output variable, or both. For each x which is both, split x into two variables, x_i and x_o for testing purposes.
4. For each local variable l, determine if l is used as an interface to the higher layer user of this protocol. If so mark l as input, output or both. Each such local variable is specifically designated, and is an input variable if it appears in an enabling predicate, and an output variable if it appears in an Action part of *predicate-action table*. If l is both input and output, split it into two variables l_i and l_o for test purposes.

3. Test Sequence Generating Procedure

Initially the test sequence is empty.

1. *state* $\leftarrow$ initial state.
2. Let $t = (p,a)$ be an untested transition from *state*.
(a) Determine the values of the input variables which make exactly one of the untested clauses of p true. Check to see if these values allow any other transition from this state to be executed. If there is one, set additional input variables to values that insure only the transition under test is enabled. Fill these in, and mark others "DC" for "don't care."
(b) Determine and mark the expected values for the output variables; also record the expected values assumed by the local variables.
(c) Set S_I to state; determine the next state and set S_E to it.

(d) Determine if S_E is transient; if not mark it as a "stop state" and skip to (3). The state is *transient* if one of its enabling predicates is true immediately upon reaching the state. This means that it can pass on to another state immediately, without waiting for further input.
(e) Attempt to make S_E into a stop state by setting "DC" values. That is, make the DC values such that, upon reaching state S_E, none of the enabling predicates are true. If successful, go to (3).
(f) If S_E is a transient state and more than one transition leaving S_E is enabled, choose one and set inputs not yet specified (if any exist), so that only one transition leaving S_E is enabled; set $t = (p,a)$ to this transition.
3. Output this test $S_I\ i_1, i_2, \ldots, i_n / o_1, o_2, \ldots, o_m\ S_E$ as the next test in the test sequence.
4. Mark the clause just tested. If all clauses in transition t are now tested, mark t as tested. If all transitions are now marked as tested, exit to "refining steps." Otherwise, continue to step (5).
5. Set state to S_E. If state is a stop state go to (2), otherwise go to step2(b).

Step 2(a) assumes that it is possible to set the input variables to values that make exactly one of the clauses true. If the protocol is well designed this assumption will generally be true. However, there is always a possibility this is not the case; if so, the test designer must choose the values so that the clauses will be tested as thoroughly as possible, perhaps in combination with other clauses. If a clause cannot be tested individually, the question of its necessity to the specification should be considered.

Step 5 sets the starting state of the next test in the sequence to the ending state of the current test. This makes the ordering of the tests follow the order of their occurrence in the actual protocol execution.

4. Refining Steps

1. Construct the I/O state diagram from the test sequence.
2. Determine if the sequence are unique, so that from each state, we have a unique input output (UIO) sequence to confirm. If not attempt to extend the sequence so that we have a unique UIO sequence from each state.
3. Check for any converging transitions. Mark these, as potential problems for testing.

The I/O diagram can be constructed from the test sequence and is a tool to help the test designer insure completeness. This finite state machine is often used as the starting point in test generation in the literature.

A UIO sequence has been defined as a sequence of inputs such that, if the input sequence is applied to the FSM when FSM is in state i, the resulting output sequence could not have been produced by the FSM when the FSM is in any other state. If the sequence of tests applied to a machine implementation in a state i is a UIO sequence, and the output is expected, then we have a stronger argument that the machine was, in fact, in state i.

This method is used by Testgen to produce the tests applied to the implementatin in this reasearch.

5. Input and Output

The protocol is specified formally as a finite state machine with local and shared variables. The FSM input text file lists the start state, the end state, the number of the transition, and the transition name. The FSM description file for T3, one of the transmitter machines of the tested implementation, is shown in Figure 15.

```
0
0 1 1 start
1 2 2 clock
2 3 3 no_data
3 1 4 delay
3 4 5 timeout1
3 4 6 timeout2
2 4 7 data
4 1 8 no_disc
4 5 9 disc
5 0 10 confirm
1 0 11 finish
```

Figure 15 : Testgen FSM Input for T3

The PAT input text file lists each transition by name, followed by a list of the predicates, followed by a list of the actions. The PAT description file for T3 is shown in Figure 16.

The test program finds all paths which may be taken through the FSM and generates a sequence of tests to check all these paths. The text output of Testgen lists all of the transitions by name and gives the expected values. For predicates, a DC indicates a ‘do not care.' For actions, a '--' indicates a 'do not care.' A example of the text output file for T3 is shown in Figure 17.

```
start      | T_active = T                          | no |
clock      | clock = T and T_active = T            | scount := inc |
no_data    | sent = F                              | count := inc |
delay      | count = LTk and scount = LTLim        | no |
timeout1   | scount = Lim                          | enqueue := T ; k := min |
timeout2   | count = k                             | enqueue := T ; k := min |
data       | sent = T                              | enqueue := T ; k := 1 |
no_disc    | scount = LTLim                        | sent := F ; count := 0 |
disc       | scount = Lim                          | sent := F ; discon := T |
confirm    | T_active = F                          | no |
finish     | T_active = F                          | no |
```

Figure 16 : Testgen PAT Input for T3

```
Trans     |   | input variables                            |**| output variables
---------------------------------------------------------------------------------
name      |Si |T_active|clock|count(i) |scount(i)|sent(i)|**|count(o)|discon|enqueue|k   |scount(o)|sent(o)|  Se|
*********************************************************************************
start     | 0 |T       |DC   |DC       |DC       |DC     |**| --     |--    |--     |--  |--       |--     |   1|
finish    | 1 |F       |DC   |DC       |DC       |DC     |**| --     |--    |--     |--  |--       |--     |   0|
clock     | 1 |T       |T    |DC       |DC       |DC     |**| --     |--    |--     |--  |inc      |--     |   2|
no_data   | 2 |DC      |DC   |DC       |DC       |F      |**| inc    |--    |--     |--  |--       |--     |   3|
delay     | 3 |DC      |DC   |LTk      |LTLim    |DC     |**| --     |--    |--     |--  |--       |--     |   1|
data      | 2 |DC      |DC   |DC       |DC       |T      |**| --     |--    |T      |1   |--       |--     |   4|
disc      | 4 |DC      |DC   |DC       |Lim      |DC     |**| --     |T     |--     |--  |--       |F      |   5|
confirm   | 5 |F       |DC   |DC       |DC       |DC     |**| --     |--    |--     |--  |--       |--     |   0|
no_disc   | 4 |DC      |DC   |DC       |LTLim    |DC     |**| 0      |--    |--     |--  |--       |F      |   1|
timeout1  | 3 |DC      |DC   |DC       |Lim      |DC     |**| --     |--    |T      |min |--       |--     |   4|
timeout2  | 3 |DC      |DC   |k        |DC       |DC     |**| --     |--    |T      |min |--       |--     |   4|
```

Figure 17 : Testgen Output for T3

B. MACHINE TEST TRACE GENERATION

Each of the eight machines was modified to open an output file and write out the values of select local and shared state variables at each point where a state transition occurred.

First, the code had to be carefully scrutinized to determine exactly where each state transition occurs. At the point immediately before leaving the current state, a call to the testdump function was made. The testdump function takes as input parameters the value of the current state and the value of the next state to be entered. This information along with the values of the key local and shared state variables is written to the trace output file.

The format of the output trace file matches that of the Testgen output file to allow for an inspection later. An example of the output generated for T3 is shown in Figure 18.

```
Si |T_Active|clock|count(i)|scount(i)|sent|**|count(o)|Discon|enqueue|  k |scount(o)|sent(o)| Se
 0 |   T    |  T  | LTk    |  LTLim   | F  |**|      0 |   F  |   F   |   1|    0    |     F | 1
 1 |   T    |  T  | LTk    |  LTLim   | F  |**|      0 |   F  |   F   |   1|   inc   |     F | 2
 2 |   T    |  T  | LTk    |  LTLim   | F  |**|    inc |   F  |   F   |   1|   ---   |     F | 3
 3 |   T    |  T  |  k     |  LTLim   | F  |**|    --- |   F  |   T   | min|   ---   |     F | 4
 4 |   T    |  T  | LTk    |  LTLim   | F  |**|      0 |   F  |   F   | ---|   ---   |     F | 1
 1 |   T    |  T  | LTk    |  LTLim   | F  |**|      0 |   F  |   F   | ---|   inc   |     F | 2
 2 |   T    |  T  | LTk    |  LTLim   | F  |**|    inc |   F  |   F   | ---|   ---   |     F | 3
 3 |   T    |  T  | LTk    |  LTLim   | F  |**|    --- |   F  |   F   | ---|   ---   |     F | 1
 1 |   T    |  T  | LTk    |  LTLim   | T  |**|    --- |   F  |   F   | ---|   ---   |     T | 2
 2 |   T    |  T  |  k     |  LTLim   | T  |**|    --- |   F  |   T   |   1|   ---   |     T | 4
 4 |   T    |  T  |  k     |  LTLim   | F  |**|      0 |   F  |   F   |   1|   ---   |     F | 1
 1 |   F    |  T  | LTk    |  LTLim   | T  |**|      0 |   F  |   F   |   1|   inc   |     T | 2
 2 |   F    |  T  | LTk    |  LTLim   | T  |**|      0 |   F  |   T   |   1|   ---   |     T | 4
 4 |   F    |  T  | LTk    |  LTLim   | F  |**|      0 |   F  |   F   |   1|   ---   |     F | 1
 1 |   F    |  F  | LTk    |  LTLim   | F  |**|      0 |   F  |   F   |   1|   ---   |     F | 0
```

Figure 18 : T3 Trace Output Example

C. THE VERIFICATION PROGRAMS

Two programs were written to perform the verification process. TestPack is an Ada package which performs comparisons between the specification and the implementation. SNRTest is an Ada program which calls TestPack and allows the user to perform up to eight different verifications with one run.

1. TestPack

Many of the trace files consisted of greater than 100 lines of output, and for some machines, such as T2, the number of predicates and actions was greater than 30. In order to check the trace files against the requirements generated by Testgen, a program called Testpack was written. The source code for Testpack can be found in the appendix.

TestPack receives a record called *file_name_record* which contains the name of the files to test. The *file_name_record* is processed to resolve the actual names and directory locations of the files to be acted upon.

TestPack starts by opening the Testgen file and parsing it. It skips everything until it locates a line starting with a '-' which indicates the next line will contain the predicate and action labels. The labels are read in and saved in an array called *names*. A counter is used to keep track of how may predicates and actions there are for this particular machine.

The program then skips down to where the requirements data is located. A temporary variable of type Testgen record is used to store the Testgen output. A Testgen record consists of the start state, the end state, the transition name, an array of strings called *values* which holds the expected predicates and actions, and *times_taken*: a counter used to generate final statistics.

Once all the data for a given line has been read, the program attempts to insert the data into the array of Testgen records called *test_states*. The transition name is checked against all those already saved in *test_states*, and if it has not been inserted yet, it is added to the list. This process continues until the Testgen output file is exhausted.

The next step in the process it to begin reading in the trace file. The program is designed to allow the user to input the source directory path as numerous directories were needed to hold the results of the different tests. The path is added to the file name and the file is opened for reading. At the same time an output file is opened in the same subdirectory to receive the results of the comparisons.

The program skips down to the first line of data and begins reading it in, first the start state, then the end state. The predicate and action values are stored in an array of strings called *temp*.

The program loops while attempting to match both the start and end state values of a saved testgen records in *test_states*. When both are matched, the test_state array index number is saved in an integer array called *a_states_match*, then the function *Match_predicates* is called. *Match_predicates* compares the values of the predicates, checking only those not listed as 'do not care' and returns a boolean indicating the result.

A predicate match success leads to the calling of the *Match_actions* function. *Match_actions* attempts to match all those actions not listed as 'do not care' and returns a boolean indicating the result.

A successful return from *Match_actions* causes the function *Report_match* to be called which writes the start and end state data followed by the transition name to the output file, see Figure 19.

```
THE MODE IS 0

match Si:  0 Se:  1  Transition: start
match Si:  1 Se:  2  Transition: clock
match Si:  2 Se:  3  Transition: no_data
match Si:  3 Se:  4  Transition: timeout2
match Si:  4 Se:  1  Transition: no_disc
match Si:  1 Se:  2  Transition: clock
match Si:  2 Se:  3  Transition: no_data
match Si:  3 Se:  1  Transition: delay
```

Figure 19 : Predicate and Action Match

Failing *Match_actions* causes the *Report_error* function to be called which indicates which action failed to match and then echoes out both the test data line and the Testgen data which had matching predicates. This information tells the user exactly where and how the implementation deviated from the specification, see Figure 20.

```
ERROR--Transition -> clock          ACTION -> scount(o) failed to match. Found: ---     Expected: inc
T_active |clock|count(i)  |scount(i) |sent(i)|count(o) |discon|enqueue |k     |scount(o) |sent(o)    |
T        |T    |LTk       |LTLim     |T      |---      |F     |F       |---   |---       |T          |
T        |T    |DC        |DC        |DC     |--       |--    |--      |--    |inc       |--         |
```

Figure 20 : Predicate Match With Action Match Failure

If all the test states have been examined without a match, *Report_error* is called which outputs the trace data and a message indicating a failure to match any of the existing transitions, see Figure 21.

```
ERROR--Predicate failed to match transition from Si:  1 to Se:  2
T_active |clock|count(i)  |scount(i) |sent(i)|count(o) |discon|enqueue |k     |scount(o) |sent(o)    |
F        |T    |LTk       |LTLim     |T      |0        |F     |F       |1     |inc       |T          |
Should have matched:
T        |T    |DC        |DC        |DC     |--       |--    |--      |--    |inc       |--         |
```

Figure 21 : Predicate Match Failure

Testpack continues reading in trace information until the data is exhausted. The final step in the automated analysis to write out the statistics concerning each transition. Every successful match caused the *times_taken* counter in the Testgen record to be incremented. The labels for the predicates and actions are written followed by the data in each of the Testgen records. The taken information allows the user to identify transitions which have not been tested, see Figure 22.

```
Taken Si Se   Name     T_active|clock|count(i)|scount(i)|sent(i)|count(o)|discon|enqueue|k  |scount(o)|sent(o)
  1    0  1   start    T       |DC   |DC      |DC       |DC     |--      |--    |--     |-- |--       |--
  1    1  0   finish   F       |DC   |DC      |DC       |DC     |--      |--    |--     |-- |--       |--
  2    1  2   clock    T       |T    |DC      |DC       |DC     |--      |--    |--     |-- |inc      |--
  2    2  3   no_data  DC      |DC   |DC      |DC       |F      |inc     |--    |--     |-- |--       |--
  1    3  1   delay    DC      |DC   |LTk     |LTLim    |DC     |--      |--    |--     |-- |--       |--
  2    2  4   data     DC      |DC   |DC      |DC       |T      |--      |--    |T      |1  |--       |--
  0    4  5   disc     DC      |DC   |DC      |Lim      |DC     |--      |T     |--     |-- |--       |F
  0    5  0   confirm  F       |DC   |DC      |DC       |DC     |--      |--    |--     |-- |--       |--
  3    4  1   no_disc  DC      |DC   |DC      |LTLim    |DC     |-0-     |--    |--     |-- |--       |F
  0    3  4   timeout1 DC      |DC   |DC      |Lim      |DC     |--      |--    |T      |min|--       |--
  1    3  4   timeout2 DC      |DC   |k       |DC       |DC     |--      |--    |T      |min|--       |--
```

Figure 22 : Final Statistics Output

2. SNRTEST

SNRTest was written to facilitate the verification process. The program, the source code for which is in the appendix, allows the user to input the names of up to eight machines as well as the path to the subdirectory where the data is stored. All test trace output files have a name of the form *machine*.out such as t1.out. The Testgen output files have names of the form *machine*res.txt, such as t1res.txt, and reside in the directory from which SNRTest is invoked.

The user is first prompted to enter the name of the machines to be tested separated by single spaces. This data is parsed and saved in a record structure called a *file_name_record*. The *file_name_record* already contains the names of the files minus the machine name prefix which is prepended. A count of how many machines will be tested is kept as the program parses the input string.

The user is then asked to input the path to the data. This information is put into each of the *file_name_records*.

Finally, SNRTest loops the number of times corresponding to the count of the machines to be tested, passing a single *file_name_record* to TestPack for processing.

D. THE TESTING

The modified protocol implementation was used to send files from one machine to another. The resulting trace files, along with the Testgen output files, were then processed using the SNRtest. The results of these examinations are contained in the next chapter.

IV. TEST RESULTS

The preceding chapter presented the methods used to generate the test trace and the specification data discussed about the program used to compare the two. This chapter contains information about what was done to generate the test data, comments about the performance of the implementation as experienced during the test data generation, comments about the general approach taken to implement the tests, and the findings based on the analysis performed on the collected data.

A. BLACK BOX VERSUS WHITE BOX TESTING METHODOLOGIES

The ideal approach to performing a verification test such as this would be as a 'black box,' that is, to test the implementation from a purely outside perspective without regard to the inner workings. This method is preferable as it focuses strictly on the input and output (predicates and actions) of the machine under test and does not require the tester to know about, or necessarily understand, the inner workings of the test subject. The tester need only provide input to the machine, observe the resultant behavior and compare it to what is specified to occur.

The testing approach taken here was more a 'white box' method. Because additions to the tested code had to be made, the tester was required to become intimately familiar with the implementation in order to correctly position data collection instructions. No less than six errors were made in the initial modifications which, until identified, made it appear the implementation was at fault.

The most common error occurred in machines which had transitions which had as an action, the changing of a value used to determine the predicate condition. Each output function was called at the end of a transition, just prior to entering the next state. In reading the ending values of the modified state variables, it appeared, in some cases, that the predicate conditions had failed. This problem was overcome by saving all the 'old' values in another variable, local to the test output function, just prior to returning to the main program. A better approach would have been to make two calls to the output file, one immediately as a state is entered and a second just prior to exiting the state.

B. LIMITATIONS

The test procedure followed here can reasonably assure the tester that all tested transitions which worked will continue to work as demonstrated. It cannot, however, be guaranteed that every machine will always terminate, that is, return to the starting state.

This particular set of machines, because of the concurrent nature of their execution, presents a special set of potential problems. Unless the hardware used to run the different machines is able to support a dedicated processor for each one, the operating system will need to make scheduling decisions. The nondeterministic nature of the operating system's scheduling algorithm may cause the behavior of the machines to vary for no apparent reason.

During the conduct of the tests there were a couple of instances where the receiver was started and allowed to stabilize, but when the transmitter was started it was not able to locate the receiver. The only conclusion which could be made from this occurrence was that the receiver machines must have been scheduled in such a way as to have created a deadlock type condition. Further analysis of the interprocess communications should be undertaken to attempt to locate the cause of this rare situation.

C. DATA COLLECTION

1. Standard/Error Free Connection Tests

The initial step was to test the receivers and transmitters during the transmission of a file on an 'error free' connection. For the majority of tests a small file, consisting of 16 lines of ascii text, was used since it was large enough to require multiple blocks be used for its transmission but not so large that thousands of lines of output were generated by the various machines. The goal was to test each transition at least once, theorizing by induction, if it works once it will work correctly any number of subsequent times.

For each of the three modes the receiver was started and then the transmitter was started and ran until completion of the file transmission. The receiver was then halted to cause the output files to be closed. The data generated for all eight machines was then moved to a separate subdirectory, corresponding to the mode tested, for later analysis.

An analysis of the data collected for the error free connections revealed numerous transitions which were not taken. The untested transitions, for the most part, represented those designed to handle unexpected or lost packets and connection failures. Additional tests were called for to test these transitions.

2. Connection Interruption Test

A larger file, consisting of 1000+ lines of ascii text, was used to test how the machines would react if the connection was interrupted. The transmission time of the file was significantly long to allow for a manual disruption of the connection.

After starting the receiver, the transmitter was started. After the connection phase was completed and transmission of the file begun, the receiver was prematurely halted. The receiver was designed to allow the user to gracefully end its run by typing in the word 'ok.' This technique caused the transmitter machines to exercise additional transitions not tested during the first phase of tests. An interruption of transmission was done for connections in all three modes.

3. Refused Connection Test

The transmitter and receiver must agree on the terms of the connection before it is made. In order to cause a connection failure due to a disagreement on the connection set up parameters, machine R2 was modified to return a different block size in its connection acknowledgment to T2 than it received in its connection request from T2. Machine T2 then found the connection to be unacceptable and exercised the *unaccept* transition and halted. The T4 machine also exercised the *unaccept* transition here.

4. Lost Packet Test

The T1 machine is responsible for sending the data packets. Within state 1, T1 takes packets from the out buffer and sends them to R1. To simulate the loss of a packet, T1 was modified, for purposes of this test only, to skip a packet during the connection and send the next one in the buffer instead.

This condition caused the receiver, in mode 2, to notify the transmitter, via the R3 *R_state* packet, of the missing data packet. The transmitter then performed a retransmit on the 'lost' packet.

With the T1 machine modified to skip a packet, connections in all three modes were made. This modification enabled the retransmit transitions to be tested in T1 and confirmed the receiver's ability to notify the transmitter about 'lost' packets.

5. Duplicate Packet Test

In order to test the receiver's ability to recognize and handle duplicate data packets, T1 was modified to send one packet twice. This test run enabled the testing of the transitions *buffer1a* and *buffer2a* in machine R1 which correspond to finding duplicate packets.

To test the T2's duplicate packet transition, *discard1*, it was necessary to temporarily modify R3 to have it send another connection ack packet. The transition *discard2*, in T2, was tested by having R3 send the same data ack packet twice.

6. Testing Timeout

R3 will only execute a *timeout2*, *disc* and *confirm* if transmitter control packets are not received for a relatively long time. To make this happen required making a temporary modification to T3 to preclude it from sending *T_state* packets and to cause it to sleep periodically for a few seconds. These changes to T3 caused R3 to execute all three of these transitions after performing 14 *timeouts*, in fact, R3 executed every transition except finish under this test condition.

7. Silent Receiver Test

To test how the transmitter would respond if the receiver failed to answer a connection request, the transmitter was started without starting the receiver. This condition was exercised for all three modes, causing the transmitter to attempt connection three times before aborting. This test case exercised the *clock, ok, timeout, quit* and *retry* turnstones in T2 and the *fail* and *disc* transition in T4.

D. GENERAL CONCEPTS

In addition to verifying individual transitions, the order in which they occur must be examined. We must ensure no states are skipped, that is, the end state of one transition must be the start state for the next transition. We also must insure all machines return to the start state, state 0, upon connection completion.

We may think of every pair of states connected by a transition to be a test case. By splitting the conditional predicates, in those transitions which have them, and creating new transitions with explicit predicates, we derive a complete set of acceptable, within specification, behavior. Verifying that every possible transition for a given machine is made according to specification, along with ensuring no unspecified transitions occur and determining the machine ends at state 0, constitutes a certification for the implementation of that machine.

E. THE TRANSMITTERS

The transmitters were tested initially, as described above, by transmitting a small file in each of the three modes. After analyzing the results, assorted special test cases were performed to test those transitions not covered by the 'normal' tests.

The normal machine configuration results are listed under the Mode0, Mode1, and Mode2 columns. Machines T1 and T3 were fully exercised by the standard tests. Machines T2 and T4 required the additional special tests in order to exercise all their transitions. These results are grouped together under the Special column and referenced by with respect to which one was used to produce what results.

The results are listed as either passed, failed, or N/T for not taken. Passed indicates the machine executed the transition according to the requirements of the specification. Failed indicates some aspect of the transition was not accomplished as required. Each failure is discussed with respect to why it occurred and what effect, if any, it had on the performance and/or service.

1. Machine T1

The T1 machine executed all *tested* transitions according to the design specifications, see Table 3. The machine demonstrated successfully the full cycle of state transitions ending back in the start state as required without making any unspecified transitions. The transition *retransmit1* was not tested due to an inability to have the machine reach a point where the retrans count exceeded the block size. Retransmission of packets was performed using the *retransmit2* transition in mode 2, and did not require any special test runs.

Se	Si	Transition	Mode 0	Mode 1	Mode 2
0	1	start	passed	passed	passed
1	1	retransmit1	N/T	N/T	N/T
1	1	retransmit2	N/T	N/T	passed
1	1	transmit_blk1	passed	N/T	N/T
1	1	transmit_blk2	N/T	passed	N/T
1	1	transmit_blk3	N/T	N/T	passed
1	2	blk_completed	N/T	passed	N/T
2	3	flow_chk1	N/T	passed	N/T
2	3	flow_chk2	N/T	N/T	passed
2	1	no_flow	passed	N/T	N/T
3	1	no_err	N/T	passed	N/T

Table 3: T1 Test Results

Se	Si	Transition	Mode 0	Mode 1	Mode 2
3	1	err_chk	N/T	N/T	passed
1	0	finish	passed	passed	passed

Table 3: T1 Test Results

2. Machine T2

T2 failed to meet the specification for five transitions: *no_flow*, *err_chk*, *no_err*, *discard1*, and *discard2*, see Table 4. The failures in each were the result of failing to dequeue a packet as an action. An examination of the implementation revealed that the dequeuing of packets was actually performed in the *rcv_state* transition, which leads to each of the others. This action was actually more efficient since it had to be done in each of the subsequent transitions regardless of which one was finally taken and did not adversely affect the machine's ability to provide the user service.

In all test cases, T2 correctly returned to the starting state at the termination of the connection.

The special case tests included the Connection Interruption Test (see page 33)[1], the Refused Connection Test (see page 33)[2], the Duplicate Packet Test (see page 34)[3], and the Silent Receiver Test (see page 35)[4].

Si	Se	Transition	Mode0	Mode1	Mode2	Special
0	1	request	passed	passed	passed	passed[1,2,3,4]
1	2	accept	passed	passed	passed	passed[1,3]
1	0	unaccept	N/T	N/T	N/T	passed[2]
1	6	clock	N/T	N/T	N/T	passed[4]
6	1	ok	N/T	N/T	N/T	passed[4]
6	7	timeout	N/T	N/T	N/T	passed[4]

Table 4: T2 Test Results

Si	Se	Transition	Mode0	Mode1	Mode2	Special
7	1	retry	N/T	N/T	N/T	passed[4]
7	0	quit	N/T	N/T	N/T	passed[4]
2	0	finish1	N/T	passed	N/T	passed[3]
2	0	finish2	passed	N/T	N/T	N/T
2	0	finish3	N/T	N/T	passed	N/T
2	0	abort	N/T	N/T	N/T	passed[1]
2	3	rcv_state	N/T	N/T	passed	passed[1,3]
3	2	discard1	N/T	N/T	N/T	failed[3]
3	2	discard2	N/T	N/T	N/T	failed[3]
3	4	update	passed	N/T	passed	passed[1,3]
4	2	no_flow	failed	N/T	N/T	failed[1,3]
4	5	flow_chk1	N/T	failed	N/T	failed[1,3]
4	5	flow_chk2	N/T	N/T	passed	passed[1,3]
5	2	no_err	N/T	N/T	N/T	failed
5	2	err_chk	N/T	failed	failed	failed

Table 4: T2 Test Results

3. Machine T3

Machine T3 performed all transitions in accordance with the design specification and demonstrated correct behavior with respect to executing no unspecified transitions and returning to the start state at the end of the connection, see Table 5. None of the special test conditions were needed to exercise all the specified transitions.

Si	Se	Transition	Mode0	Mode1	Mode2
0	1	start	passed	passed	passed

Table 5: T3 Test Results

Si	Se	Transition	Mode0	Mode1	Mode2
1	2	clock	passed	passed	passed
2	3	no_data	passed	passed	passed
3	1	delay	passed	passed	passed
3	4	timeout1	N/T	N/T	passed
3	4	timeout2	passed	passed	passed
2	4	data	passed	passed	passed
4	1	no_disc	passed	passed	N/T
4	5	disc	N/T	N/T	passed
5	0	confirm	N/T	N/T	passed
1	0	finish	passed	passed	N/T

Table 5: T3 Test Results

4. Machine T4

Machine T4 executed all transitions according to specification, see Table 6, with one minor discrepancy noted below. At the end of each connection, the machine correctly returned to the start state.

A condition arose during the transmission of a large file which indicated the possible need for an additional transition definition. While the out buffer has room, the machine writes packets into it, the *write* transition in state 2. If the out buffer is full, the machine loops in state 2 waiting for room to open up. This out buffer full condition caused an apparent failure in the *write* transition a few times during the transmission of a large file when, in fact, an undefined transition was occurring. Adding a *wait_buf* transition from state 2 to state 2 would account for this situation and would better define the protocol.

The special tests included the Connection Interruption Test (see page 33)[1], the Refused Connection Test (see page 33)[2], and the Silent Receiver Test (see page 35)[3].

Si	Se	Transition	Mode0	Mode1	Mode2	Special
0	1	signal	passed	passed	passed	passed[2,3]
1	0	fail	N/T	N/T	N/T	passed[3]
1	0	unaccept	N/T	N/T	N/T	passed[2]
1	2	start	passed	passed	passed	passed[2,3]
2	2	write	passed	passed	passed	N/T
2	3	finish	passed	passed	passed	N/T
3	0	confirm	N/T	passed	passed	N/T
2	0	disc	N/T	N/T	N/T	passed[1]

Table 6: T4 Test Results

F. RECEIVERS

The receivers were tested in much the same way as the transmitters. The first set of tests involved sending a small file in each of the three modes.

As with the transmitters, the normal machine configuration results are listed under the Mode0, Mode1, and Mode2 columns. All machines required some additional special test connections in order to exercise all their transitions. These results are grouped together under the Special column and referenced by with respect to which one was used to produce what results.

The results are listed as either passed, failed, N/A for not applicable, or N/T for not taken.

1. Machine R1

The R1 machine made no unspecified transitions and correctly returned to the start state upon completion of each connection but failed all but the *start* and *receive* transitions.

The *no_buf, buffer1a, buffer1b, buffer2a,* and *buffer2b* transitions all failed to meet the action requirement of dequeuing a packet. The dequeuing of packets was performed as an action in the *receive* transition which proceeds all five listed failures. This change is actually a more efficient way to perform the dequeuing of packets and does not adversely affect the ability of the receiver to provide service to the user.

In all tests, the machine performed the *finish* transition while the *INBUF* still contained packets, see Table 7. As noted in [12], this was an error in the specification. Since R1 has the duty of servicing the *T_chan*, it makes more sense to enable the *finish* transition when the *T_chan* is empty and *R_active* is false, which is exactly what was done in the implementation. Although the transition was reported as a failure, due to a predicate mismatch, the machine did return to state 0 from state 1 as required.

The special test performed specifically to check *buffer1b* and *buffer2b* was the Duplicate Packet Test (see page 34).

Si	Se	Transition	Mode0	Mode1	Mode2	Special
0	1	start	passed	passed	passed	passed
1	0	finish	failed	failed	failed	failed
1	2	receive	passed	passed	passed	N/T
2	1	no_buf	N/T	failed	N/T	N/T
2	1	buffer1a	failed	N/T	N/T	failed
2	1	buffer1b	N/T	N/T	N/T	failed
2	1	buffer2a	N/T	N/T	failed	failed
2	1	buffer2b	N/T	N/T	N/T	failed

Table 7: R1 Test Results

2. Machine R2

Analysis of the R2 trace data confirmed the machine made no unspecified transitions and finished in the start state as required. Transitions made which did not follow the specification are noted as are some which did not occur at all.

Discard3 was not actually implemented in the code. The author Wan, [12], treated *T_state* packets, as well as *Conn_disc* packets, as out-of-band and thus they were never queued in the *T_chan* and did not need to be dequeued. This implementation change was correctly identified by the testing process when the failure to correctly execute *discard3* was discovered, see Table 8.

Start2 and *update* were both dependant on having a *T_state* packet in the *T_chan* which did not occur for the same reason: it is out-of-band. Since *T_chan* never contains a *T_state* packet, *start2* never appears to occur.

By examining the actions taken when the verification program reported the *update* transition failed the predicate condition, it was discovered that the correct action of setting *high* to equal the *T_chan(front).seq* was actually accomplished. We may consider this a success as far as performing the actions are concerned and conclude proper operation was maintained.

The *T_state* data was still read and used and the machine appeared to operate as required although it did deviate from the given design specification.

The special test performed to check *buffer1b* and *buffer2b* was the Duplicate Packet Test (see page 34)

Si	Se	Transition	Mode0	Mode1	Mode2	Special
0	1	ack	passed	passed	passed	passed
1	3	clock	N/T	N/T	N/T	passed
3	1	ok	N/T	passed	N/T	passed

Table 8: R2 Test Results

Si	Se	Transition	Mode0	Mode1	Mode2	Special
3	0	timeout	N/T	N/T	N/T	passed
1	2	start1	passed	passed	passed	passed
1	2	start2	N/T	N/T	N/T	N/T
1	2	start3	N/T	N/T	N/T	passed
2	0	finish1	passed	N/T	N/T	passed
2	0	finish2	N/T	passed	passed	passed
2	2	update	N/T	N/T	N/T	failed
2	2	discard1	N/T	N/T	N/T	passed
2	2	discard2	N/T	N/T	N/T	passed
2	2	discard3	failed	N/T	failed	passed
1	1	lost_ack	N/T	N/T	N/T	passed

Table 8: R2 Test Results

3. Machine R3

All transitions in machine R3 were performed in accordance with the specification, no unspecified transitions were made, and all connections were terminated with the machine back in the start state, see Table 9. The conditions of the special test are detailed in Testing Timeout (see page 34).

Si	Se	Transition	Mode0	Mode1	Mode2	Special
0	1	start	passed	passed	passed	passed
1	2	clock	passed	passed	passed	passed
2	3	no_data	N/T	passed	passed	passed
3	1	delay	N/T	passed	passed	passed
3	4	timeout1	N/T	passed	passed	passed

Table 9: R3 Test Results

Si	Se	Transition	Mode0	Mode1	Mode2	Special
3	4	timeout2	N/T	N/T	N/T	passed
2	4	data	passed	passed	passed	passed
4	1	no_disc	passed	passed	passed	passed
4	5	disc	N/T	N/T	N/T	passed
5	0	confirm	N/T	N/T	N/T	passed
1	0	finish	passed	passed	passed	N/T

Table 9: R3 Test Results

4. Machine R4

Machine R4 does not operate in mode 0. All transitions in machine R4 were made in compliance with the specification, see Table 10, and the machine returned to the start state at the end of each connection having taken only specified transitions. The *wait* transition was not tested due to an inability to satisfy the predicate conditions. Regardless of the modifications to the various machines, the *wait* condition was never satisfied.

The special test performed to check *disc* was the Lost Packet Test (see page 34).

Si	Se	Transition	Mode 0	Mode 1	Mode 2	Special
0	1	start	N/A	passed	passed	passed
1	0	finish	N/A	N/T	passed	N/T
1	0	disc	N/A	N/T	N/T	passed
1	2	accept1	N/A	passed	N/T	N/T
1	2	accept2	N/A	N/T	passed	passed
2	3	no_err	N/A	N/T	N/T	passed
3	1	wait	N/A	N/T	N/T	N/T
3	1	retrieve	N/A	passed	N/T	N/T

Table 10: R4 Test Results

Si	Se	Transition	Mode 0	Mode 1	Mode 2	Special
2	1	err_chk	N/A	N/T	passed	passed

Table 10: R4 Test Results

G. FINAL OBSERVATIONS

With the exception of the two failures of the transmitter to find the running receiver, the implementation appeared to perform quite well. The only comment about the protocol in general which bears mentioning concerns the parallel execution design. There is no question that parallel processing affords a great potential for reducing packet processing overhead with respect to time. Without a multiple processor hardware configuration however, this advantage is not only lost, but overhead is added due to the requirement for context switching. And of course, there is the non deterministic processor scheduling problem as well. Final conclusions and topics for further research follow in the next chapter.

V. CONCLUSIONS AND TOPICS FOR FUTURE RESEARCH

A. IMPLEMENTATION CONFORMANCE VERIFICATION

The goal of this work was to take a formal specification and an actual implementation and perform an examination to determine the implementation's compliance with the specification. The formal specification was presented, the methods for producing the set of test sequences from the specification and for generating an implementation run time trace file were presented, and the methodology for performing the comparison tests was explained. Finally, the results of the tests were presented along with the findings.

1. Testing Methods and Problems

a. Errors

Ideally, the testing of software should be as uninvasive as possible. Modification of the original source code opens the tester up for a multitude of potential problems due to the possible introduction of errors.

How does a tester distinguish the source of apparent errors? Is the error due to the original implementation, the modifications made to allow for the test, or perhaps an interaction between the two? These very problems arose during this research. The code inserted to write out the state variables incorrectly reported some values in come cases. A careful analysis of the added code was required to identify the apparent failure of the implementation as a misreported state variable value.

b. Comparisons

The problem of performing comparisons lies in matching the requirements variables to the implementation output variables. The approach taken in this work was to match the two sets of data positionally, that is, in the actual order they are found in the two data files.

Another possible approach would be to match the variable names. This would require the tester to insure exact variable name matches. This complicates the comparison process, requiring the comparison program to perform the additional task of processing the entire list of requirements variable names for each test variable in every line of output.

c. *Complete Testing*

The goal of any test is to verify every possible case. Two transitions out of a total of 96 were not tested due to an inability to cause the target machines to make them. A number of transitions only occur when an error is encountered, either due to a lost or unexpected packet or due to a connection failure. Simulating these cases required modifying some of the machines to either purposely produce bad packets or to loose packets entirely. Additionally, manually interrupting the execution of a set of machines during an actual connection was done to simulate a connection failure.

B. CONCLUSIONS

The implementations varied slightly from the formal specification. An examination of the modifications made indicates the variations do not alter the intended behavior of the machines from an outside or blackbox perspective.

Formal specifications rarely anticipate all possible actual implementation situations. Methods for optimizing performance and providing for unanticipated situations may require that the formal specification be amended. The software development cycle almost always requires the modification of specifications. The variations from the given specification identified in this implementation examination represent just such a situation.

C. TOPICS FOR FOLLOW ON WORK

1. A Formal Approach to Implementing a Formal Specification

Given a formal specification in the form of SCM, a formal approach to generating the implementation should be used. The specification describes the protocol's behavior using a FSM. A structured approach to writing the implementation using case statements corresponding to the given FSM states would greatly enhance the readability and understandability of the code and make it much easier to test. Additionally, the variable names given in the specification should match exactly.

The implementation would consist of a set of blocks. Each state, represented as a case, would contain conditional statements corresponding to each possible transition from that state. The conditions for executing a transition would be taken directly from the specification's predicate requirements. The body of these transition statements would then contain the actions required by the specification followed by an update of the case variable to reflect the next state to be entered.

The test bench program, proposed above, would be able to trace the state transitions by monitoring the value of the case state variable. A change in the value would be a cue to sample the machine's state variables and write them to the trace file along with the data on the initial state and end state for the transition.

The implementations examined here were, to some degree, designed in this fashion. With minor modifications, the existing code could be modified to follow the proposed style.

2. Examining an Implementation Unintrusively

To externally manipulate and examine an implementation would be preferable. Having a test bench on which to perform the test runs would be ideal. A master program to act as a wrapper through which all machine input and output would have to pass would facilitate both the development and the testing processes. This master program would also be the place to embed the test run output generation program.

Examining and recording the values of internal state variables during a program's execution from the outside would eliminate the problem of introducing errors into the implementation. Additionally, it would allow the tester more freedom to examine different variations of the implementation. The method of modifying the code of the implementation requires a great deal of time on the part of the tester to read and understand the code being tested as well as to find the appropriate places to insert the test code to write out the machine's state and variable values.

APPENDIX. RESULTS ANALYSIS GENERATING PROGRAM

A. SNRTEST.A

```
with text_io, Test_Snr;
use text_io, Test_Snr;

procedure SNRTest is

 package integer_inout is new integer_io(integer);
 use integer_inout;

subtype name is string(1..2);

myfiles              : array(0..9) of file_name_record_type;
instring             : string(1..30) := (others => ' ');
instring_len         : integer:=1;
Number_to_test       : integer := 1;
indx                 : integer := 1;
snames                : array(0..7) of name;

begin

  text_io.put_line("What machines are we testing? ");
  text_io.get_line(instring, instring_len);

  for I in 0..7 loop
    for II in 1..2 loop
      names(I)(II) := instring(Indx);
      indx := indx + 1;
    end loop;
    indx := indx + 1;
    exit when instring(indx) = ' ';
    Number_to_test := Number_to_test + 1;
  end loop;

for J in 0..Number_to_test-1 loop
  for I in 1..2 loop
    myfiles(J).File1(I):= names(J)(I);
    myfiles(J).File2(I):= names(J)(I);
    myfiles(J).File3(I):= names(J)(I);
  end loop;
end loop;

 text_io.put_line("What subdirectory contains the test output?");
 text_io.get_line(instring, instring_len);

for J in 0..Number_to_test loop
 for I in 1..instring_len loop
   exit when instring(I) = ' ';
```

```
      myfiles(J).path(I) := instring(I);
    end loop;
  end loop;

  for I in 0..Number_to_test-1 loop
    tgparse(myfiles(i));
  end loop;

  end SNRTest;
```

B. TESTPACK.A

```
-- Thesis: Testing the SNR Transport Protocol
--
-- Captain Bob Grier
--
-- Package Specification and Body for 'Test_Snr'
--
-- Purpose: Automated comparison of protocol specification and test run
--          results.
--
-- Description:  The package takes a record containing the names of
--          specification file: *res.txt, the test output file : *.out
--          and the name of the file to write the results of the
--          comparison to: *test.rst.
--             The specification is read and each unique transition is saved
--          into a data structure containing the start and end states as well
--          as the predicate and action values required.
--             The test output file is then read one line at a time. The start
--          and end states are compared against the saved specification records
--          until a match is found. The predicates are first checked to see if
--          they match, a failure to match generates an error message indicating
--          the predicates were not able to be matched and lists all transitions
--          which start and end the same as the test transition. The user can
--          check which predicate fields are not correct by comparing the actual
--          list of predicates with the expected predicates.
--             Next, the actions are checked. A failure to match all the actions
--          results in an error message listing which action failed to match as
--          well as what was found and what was expected. The test data and the
--          transition are listed to allow for further comparison.
--             Matches are listed showing the start and end states and the name
--          of the transition taken.
--
-------------------------------------------------------------------------

with text_IO;
use text_IO;

package Test_Snr is
```

```
  type file_name_record_type is record
    File1 : string(1..13) := "  res.txt    ";
    File2 : string(1..13) := "  .out       ";
    File3 : string(1..13) := "  test.rst   ";
    Path  : string(1..30) := (others =>' ');
  end record;

  procedure tgparse (my_files: in file_name_record_type);

end Test_Snr;

package body Test_Snr is

  procedure tgparse(my_files: in file_name_record_type) is

    test_state_number : constant integer := 35;

    subtype result is string(1..15);
    type the_results is array(1..35) of result;

    type testgen_record_type is record
      Si                  : integer := -1;
      Se                  : integer := -1;
      T_name              : string(1..15);
      values              : the_results;
      times_taken         : integer := 0;
    end record;

    test_states           : array(1..Test_state_number) of testgen_record_type;

    temp,
    names                 : the_results := (others =>(others=>' '));

    a_line                : string(1..300) := (others => ' ');
    snipit                : string(1..10) := (others => ' ');
    t_name                : string(1..15);
    This_char             : character;
    full_in_name,
    full_out_name         : string(1..50) := (others => ' ');

    a_line_len            : natural;
    dummy                 : positive;
    how_many,
    Si,
    sm_index,
    Se,
    indx,
    ti,
    ri,
    bar_count             : integer := 1;
    predicate_count       : integer := -2;
    a_states_match          : array(1..6) of integer;

    predicate               : Boolean := True;

 -- FILES
    testgen_file,
    out_file,
    TEST_Ofile              : File_type;

  package integer_inout is new integer_io(integer);
  use integer_inout;

--------------------------------------------------------------------------
-- FUNCTION: LENGTH -- finds the length of a string
--------------------------------------------------------------------------

function length(mystring: in string) return integer is
   len                      : integer:=0;
 begin
   for I in 1..100 loop
     exit when mystring(i) = ' ';
     len := len + 1;
   end loop;
   return len;

 end length;

--------------------------------------------------------------------------
-- PROCEDURE: OPEN_THE_FILE -- opens the input files
--------------------------------------------------------------------------

procedure OPEN_THE_FILE(FILE : in out FILE_TYPE; File_name: in string) is
 THE_FILE                    : FILE_TYPE;
 FILE_NAME_LENGTH        : INTEGER :=0;

 begin
   FILE_NAME_LENGTH := length(File_name);
   OPEN(FILE, MODE=> IN_FILE, NAME=>FILE_NAME(1..FILE_NAME_LENGTH));
   NEW_LINE;

 end OPEN_THE_FILE;

--------------------------------------------------------------------------
-- PROCEDURE: OPEN_OUT_FILE -- opens the output file
--------------------------------------------------------------------------

procedure OPEN_OUT_FILE(FILE : in out FILE_TYPE; File_name: in string) is
 THE_FILE                    : FILE_TYPE;
 FILE_NAME_LENGTH        : INTEGER :=0;

 begin
```

```
    FILE_NAME_LENGTH := length(File_name);
    create(FILE, NAME=>FILE_NAME(1..FILE_NAME_LENGTH));
    NEW_LINE;
  end OPEN_out_FILE;

-----------------------------------------------------------------------------
-- PROCEDURE: print_names -- writes out the predicate and action names
-----------------------------------------------------------------------------

procedure print_names is

begin
  for II in 3..how_many+2 loop
    put(out_file,names(II)(1..10));
    put(out_file,"|");
  end loop;
  new_line(out_file);

end print_names;

-----------------------------------------------------------------------------
-- PROCEDURE: print_transition -- prints out the given transition
-----------------------------------------------------------------------------

procedure print_transition (Transition: in out integer) is

begin
 for II in 1..how_many loop
    put(out_file,test_states(Transition).values(II)(1..10));
    put(out_file,"|");
  end loop;
  new_line(out_file);

end print_transition;

-----------------------------------------------------------------------------
-- PROCEDURE: Report_Match -- prints out the match condition
-----------------------------------------------------------------------------

procedure Report_Match(Transition: in integer) is

begin

 test_states(Transition).times_taken := test_states(Transition).times_taken + 1;
 put(out_file,"match Si: “);
 put(out_file,Si,2);
 put(out_file," Se: “);
 put(out_file,Se,2);
 put(out_file,"  Transition: “);
 put(out_file,test_states(Transition).T_name);
 new_line(out_file);
```

```
end Report_Match;

---------------------------------------------------------------------------
-- PROCEDURE: Report_error -- reports falure to match any transitions
---------------------------------------------------------------------------

procedure Report_error is  -- this is the no predicate match case

begin

  new_Line(out_file);
  put(out_file,"ERROR--Predicate failed to match transition from Si: ");
  put(out_file,test_states(a_states_match(1)).Si,2);
  put(out_file," to Se: ");
  put(out_file,test_states(a_states_match(1)).Se,2);
  new_line(out_file);
  print_names;

  for II in 2..how_many+1 loop
    put(out_file,temp(II)(1..10));
    put(out_file,"|");
  end loop;
  new_line(out_file);
  put(out_file,"Should have matched:");
  new_line(out_file);
  for II in 1..6 loop
    exit when a_states_match(II) = -1;
    print_transition(a_states_match(II));
  end loop;
  new_Line(out_file);

end report_error;

---------------------------------------------------------------------------
-- PROCEDURE: Report_error -- reports falure to match actions when predicate
--                            is matched
---------------------------------------------------------------------------

procedure Report_error (Transition: in integer;
                        Predicate: in integer) is

  trans_out    : integer := Transition;

begin
  new_line(out_file);
  put(out_file,"ERROR--Transition -> ");
  put(out_file,test_states(Transition).T_name);
  put(out_file,"ACTION -> ");
  put(out_file,names(Predicate+2)(1..10));
  put(out_file,"failed to match.");
```

```
    put(out_file," Found: ");
    put(out_file, temp(Predicate+1));
    put(out_file," Expected: ");
    put(out_file,test_states(Transition).values(Predicate));

    new_line(out_file);
    print_names;
    for II in 2..how_many+1 loop
      put(out_file,temp(II)(1..10));
      put(out_file,"|");
    end loop;
    new_line(out_file);
    print_transition(trans_out);
    new_line(out_file);

  end report_error;

  ----------------------------------------------------------------------------
  -- FUNCTION:  Match_predicates -- attempts to match the predicates
  ----------------------------------------------------------------------------

  function Match_predicates(Transition: integer) return boolean is

    DC1             : result := "DC           ";

  begin

    for K in 1..predicate_count loop
      if test_states(Transition).values(k) /= DC1 then
        if test_states(Transition).values(k) /= temp(k+1) then
          return False;
        end if;       -- end of the predicate/action missmatch condition
      end if;     -- end of the check all non-don't care values
    end loop;  -- end of "check all predicates and actions" loop
    return True;

  end Match_predicates;

  ----------------------------------------------------------------------------
  -- FUNCTION:  Match_actions -- attempts to match the actions
  ----------------------------------------------------------------------------

  function Match_actions(Transition: integer) return boolean is

    DC2             : result := "--           ";
    out_trans        : integer := Transition;
  begin

    for K in predicate_count+1..how_many loop
      if test_states(Transition).values(k) /= DC2 then
        if test_states(Transition).values(k) /= temp(k+1) then
        Report_error(out_trans,k);
        return False;
      end if;      -- end of the predicate/action missmatch condition
    end if;    -- end of the check all non-don't care values
  end loop;  -- end of "check all predicates and actions" loop
  return True;

end Match_actions;

--------------------------------------------------------------------------
-- Main body
--------------------------------------------------------------------------

begin
  open_the_file(testgen_file, my_files.file1);

  get(testgen_file, this_char);

  while This_char /= '-' loop     -- skip forward to data lines
    get(testgen_file, this_char);
    skip_line(testgen_file);
  end loop;

  get_line(testgen_file, a_line, a_line_len);

  ti := 1;
  ri := 1;

  for I in 1..a_line_len loop          -- find the predicates and actions
    if a_line(i) /= ' ' and a_line(i) /= '|' and a_line(i) /= '*' then
      names(ti)(ri) := a_line(i);
      ri := ri + 1;
    elsif a_line(i) = '|' and a_line(i-1) /= '*' then -- ignore the |**|
      if predicate then
        predicate_count := predicate_count + 1;
      end if;
      ri := 1;
      ti := ti + 1;
    end if;
    if a_line(i) = '*' then
      predicate := false;
    end if;
  end loop;
put("prediate count is: "); put(predicate_count, 3); new_line;
 skip_line(testgen_file);

 While not End_of_file(testgen_file) loop

  get_line(testgen_file, a_line, a_line_len);

-- FIRST GET THE STATE NUMBERS --
```

```
    indx := 1;
    T_name:= (others =>' ');

    while a_line(indx) /= '|' loop  -- read until the first bar '|'
      t_name(indx) := a_line(indx); -- Si is after this bar
      indx := indx + 1;
    end loop;


  for I in 1..10 loop              -- put the next 10 characters into snipit
    snipit(i) := a_line(indx+i);
  end loop;
  integer_inout.get(snipit, Si, dummy);  -- read the start state as an integer

  for I in 1..3 loop                  -- pick the end state off the end
     if a_line(i+a_line_len-3) = '|' then
       snipit(i) := ' ';
     else
       snipit(i) := a_line(i+a_line_len-3);
     end if;
  end loop;

  integer_inout.get(snipit, Se, dummy);  -- read the end state as an integer

-- reset values

  ti := 1;
  ri := 1;
  bar_count := 0;
  temp := (others =>(others => ' '));

  for I in 1..a_line_len loop           -- find the predicates and actions
    if a_line(i) /= ' ' and a_line(i) /= '|' and a_line(i) /= '*' then
      temp(ti)(ri) := a_line(i);
      ri := ri + 1;
    elsif a_line(i) = '|' and a_line(i-1) /= '*' then -- ignore the |**|
      ri := 1;
      ti := ti + 1;
    end if;

    if a_line(i) = '|' then
      bar_count := bar_count + 1;
    end if;

  end loop;

-- attempt to insert into the Test_state array
-- bar count minus 4 is the actual number of predicates and actions

 how_many := bar_count - 4;
```

```
  for I in 1..Test_State_number loop        -- insert new data loop
   if test_states(i).T_name = T_name then   -- check for repeats
     exit;
   elsif test_states(i).Si = -1 then        -- if we haven't seen this one...
     test_states(i).T_name := t_name;       -- add this one onto the list
     for j in 1..how_many loop              -- add the states
       test_states(i).values(j) := temp(j+2); -- don't want the first two fields
     end loop;
     test_states(i).Si := Si;             -- add the Si info
     test_states(i).Se := Se;               -- add the Se info
     exit;
   end if;
  end loop;    -- insert new data loop
end loop;  -- end of reading the testgen file

--------------------------------------------------------------------------
--  read and check the test output against the testgen stuff
--------------------------------------------------------------------------

 indx := 1;                    -- add the path if any
 for I in 1..30 loop
   exit when my_files.Path(I) = ' ';
   full_in_name(indx) := my_files.path(I);
   full_out_name(indx) := my_files.path(I);
   indx := indx + 1;
 end loop;

 for I in 1..13 loop
   full_in_name(indx) := my_files.file2(I);
   full_out_name(indx) := my_files.file3(I);
   indx := indx + 1;
 end loop;

 open_the_file(test_ofile, full_in_name); --my_files.file2
 open_out_file(out_file, full_out_name); --  my_files.file3
 skip_line(test_ofile);

 put(out_file,"THE MODE IS ");
 put(out_file,my_files.Path(2));
 new_line(out_file,2);
 while not End_of_file(test_ofile) loop
  get_line(test_ofile, a_line, a_line_len);

  integer_inout.get(a_line, Si, dummy);  -- get Si

  for I in 1..3 loop
    if a_line(i+a_line_len-3) = '|' then
      snipit(i) := ' ';
    else
      snipit(i) := a_line(i+a_line_len-3);
    end if;
   end loop;

   integer_inout.get(snipit, Se, dummy);  -- get Se

   ti := 1;
   ri := 1;
   bar_count := 0;
   temp := (others=>(others=>' '));

   for I in 1..a_line_len loop  -- find the predicates and actions
     if a_line(i) /= ' ' and then
        a_line(i) /= '|' and then
        a_line(i) /= '*' then

       temp(ti)(ri) := a_line(i);
       ri := ri + 1;
     elsif a_line(i) = '|' and then a_line(i-1) /= '*' then
       ri := 1;
       ti := ti + 1;
     end if;

     if a_line(i) = '|' then
       bar_count := bar_count + 1;
     end if;

  end loop;  -- end of parsing this line

--- the loading of the test stuff goes here **************************

  sm_index := 1;
  a_states_match := (others=> -1);

  for I in 1..Test_State_number loop

   if test_states(i).Si = -1 then
     Report_Error;
     exit;
   end if;

   if test_states(i).Si = Si and then test_states(i).Se = Se then
     a_states_match(sm_index):= I;
     sm_index := sm_index +1;
     if Match_predicates(I) then
       if Match_actions(I) then
         Report_match(I);
         exit;
       else
         exit;
       end if;
     end if;
```

```
        end if;

      end loop; -- end of the "check all the test states" loop

    end loop;  -- read the test output file

  -- echo out the specifiations
    new_line(out_file);
    put(out_file,"#  Taken Si  Se   Name        ");
    for II in 3..how_many+2 loop
      put(out_file,names(II)(1..10)); put(out_file,"|");
    end loop;
    new_line(out_file);

    for I in 1..Test_State_number loop
      exit when test_states(i).Si = -1;
      put(out_file,I,2);
      put(out_file,test_states(i).times_taken, 4);
      put(out_file,test_states(i).Si, 4);
      put(out_file,test_states(i).Se, 4);
      put(out_file,"  ");
      put(out_file,test_states(i).T_name(1..14));
      for j in 1..how_many loop
        put(out_file,test_states(i).values(j)(1..10));
        put(out_file,"|");
      end loop;
      new_line(out_file);
    end loop;

  end tgparse;

  end Test_Snr;
```

LIST OF REFERENCES

[1] Netravali, A., Roome, W., and Sabnani, K., "Design and Implementation of a High Speed Transport Protocol," *IEEE Transactions in Communications,* vol.38, #11, Nov 1990.

[2] Lundy, G.M., McArthur, R.C., "Formal Modal of a High Speed Transport Protocol," *Protocol Specification, Testing and Verification* XII, North-Holland, 1992.

[3] Brinksma, E., "A tutorial on LOTOS," Proc IFIP WG 6.1 5th Int Workshop on Protocol Specification, Testing and Verification, Toulouse-Moissac, France, June 10-13, 1985.

[4] Castenet, R., Dupuex, A., Guitton, P., "Ada, a Well-suited Language for the Specification and Implementation of Protocols," Proc IFIP WG 6.1 5th Int Workshop on Protocol Specification, Testing and Verification, Toulouse-Moissac, France, June 10-13, 1985.

[5] Budkowsky, S., Dembinsky, P., "The Formal Specification Technique Estelle," *Computer Networks and ISDN* Syst Vol. 14, 1987.

[6] Diaz, M., Ansart, J.P., Courtiat, J., Azema, P., Chari, V., *The Formal Description Technique Estelle,* North-Holland Elvisier, 1989.

[7] Linn, R.J., "The Features and Facilities of Estelle: a Formal Description Technique Based upon an Extended Finite State Machine Model," Proc IFIP WG 6.1 5th Int Workshop on Protocol Specification, Testing and Verification, Toulouse-Moissac, France, June 10-13, 1985.

[8] Holzmann, G. J., *Design and Validation of Computer Protocols,* Prentice Hall Software Series, 1991.

[9] Lundy, G.M., Miller, R.E., "Specification and Analysis of a Data Transfer Protocol Using Systems of Communicating Machines," *Distributed Computing,* December 1991.

[10] Tipici, H.A., "Specification and Analysis of a High Speed Transport Protocol," M.S. Thesis, Naval Postgraduate School, Monterey, CA., 1993.

[11] Lundy, G. M., Basaran, C, "Automated Generation of Protocol Test Sequences From Formal Specifications," February 1994.

[12] Wan, W. J., "Implementation of the SNR High-Speed Transport Protocol (the Receiver Part)," MS Thesis, Naval Postgraduate School, Monterey, CA., March 1995.

[13] Mezhoud, F., "Implementation of the SNR High-Speed Transport Protocol (the Transmitter Part)," MS Thesis, Naval Postgraduate School, Monterey, CA., March 1995.

[14] Miller, R.E., Lundy, G.M., "Testing Protocol Implementations Based on a Formal Specification," *Protocol Test Systems III,* North-Holland, 1990.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22304-6145

Dudley Knox Library 2
Code 052
Naval Postgraduate School
Monterey, CA 93943- 5101

Chairman, Code CS 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Dr G. M. Lundy, Code CS/Lu 2
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Dr Shridhar B. Shukla, Code, EC/Sh 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

CPT Robert B. Grier, Jr. 1
904 Tomahawk Trail
Kerrville , TX 78028